# 目錄

# Hyperledger 源码分析之 Fabric

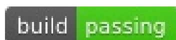


图 1.1.1 - Build Status

区块链技术是计算机技术与金融技术交融的成功创新，被认为是极具潜力的分布式账本平台的核心技术。如果你还不了解区块链，可以阅读 区块链技术指南。

作为 Linux 基金会支持的开源分布式账本平台，Hyperledger 受到了众多企业的支持和开源界的关注。本书将试图剖析 Hyperledger Fabric 项目相关源码，帮助大家深入理解其实现原理。

本书适用于对区块链技术和超级账本 Fabric 项目有一定实践经验，并对其实现感兴趣的读者。

- 在线阅读：GitBook 或 GitHub。
- pdf 版本 下载
- epub 版本 下载

Hyperledger Fabric 自身仍在快速发展中，生态环境也在蓬勃成长。欢迎 参与维护项目。

- 修订记录
- 贡献者名单

# 进阶学习




图 *1.1.2* - 区块链原理、设计与应用

《区块链原理、设计与应用》已经正式出版，详细介绍了区块链相关技术，特别超级账本项目的设计、架构和应用，欢迎大家阅读使用并反馈建议。

- 京东图书

- China-Pub
- 当当图书

# 鼓励项目

欢迎鼓励项目一杯 coffee~



图 *1.1.3 - coffee*

# 技术交流

欢迎大家加入 区块链技术讨论群 。

- QQ 群 I：335626996（已满）
- QQ 群 II：523889325（已满）
- QQ 群 III：414919574（已满）
- QQ 群 IV：364824846（可加）

本书结构由 *gitbook_gen* 维护。

# 版本历史

- 0.7.0: 2017-XX-YY
    - 添加核心过程分析。
- 0.6.0: 2017-10-01
    - configtxgen 模块；
    - cryptogen 模块；
    - 根据 fabric 1.0 调整结构。
    - 根据 fabric 1.0.1 调整结构。
- 0.5.0: 2017-05-08
    - 完成 peer 模块；
    - 按照最新代码结构更新；
    - 完成部分 orderer 模块分析。
- 0.4.0: 2016-12-12
    - 完成 0.6 分支；
    - 开始 1.0 分支新架构代码。
- 0.3.0: 2016-08-04
    - 完成主要模块内容。
- 0.2.0: 2016-07-01
    - 基本功能分析。
- 0.1.0: 2016-06-08
    - 完成基础框架。

# 参与贡献

贡献者 名单。

本书源码开源托管在 Github 上，欢迎参与维护：github.com/yeasy/hyperledger_code_fabric。

首先，在 GitHub 上 `fork` 到自己的仓库，如 `docker_user/hyperledger_code_fabric`，然后 `clone` 到本地，并设置用户信息。

```
$ git clone git@github.com:docker_user/hyperledger_code_fabric.git
$ cd hyperledger_code_fabric
$ git config user.name "yourname"
$ git config user.email "your email"
```

更新内容后提交，并推送到自己的仓库。

```
$ #do some change on the content
$ git commit -am "Fix issue #1: change helo to hello"
$ git push
```

最后，在 GitHub 网站上提交 pull request 即可。

另外，建议定期使用项目仓库内容更新自己仓库内容。

```
$ git remote add upstream https://github.com/yeasy/hyperledger_code_fabric
$ git fetch upstream
$ git checkout master
$ git rebase upstream/master
$ git push -f origin master
```

# 整体结构

Hyperledger Fabric 在 1.0 中，架构已经解耦为三部分：

- fabric-peer：主要起到 peer 作用，包括 endorser、committer 两种角色；
- fabric-ca：即原先的 membersrvc，独立成一个新的项目。
- fabric-order：起到 order 作用。

其中，fabric-peer 和 fabric-order 代码暂时都在 fabric 项目中，未来可能进一步拆分。

## 核心代码

fabric 项目中主要包括代码、工具、脚本等部分，核心源代码目前约为 430 个文件，80K 行。

```
$ cd fabric
$ find bccsp common core events gossip msp orderer peer protos \
    -not -path "*/vendor/*" \
    -name "*.go"  \
    -not -name "*_test.go" \
    | wc -l
431
$ find bccsp common core events gossip msp orderer peer protos \
    -not -path "*/vendor/*" \
    -name "*.go"  \
    -not -name "*_test.go" \
    | xargs cat | wc -l
80560
```

### 源代码

实现 fabric 功能的核心代码，包括：

- bccsp 包：实现对加解密算法和机制的支持。
- common 包：一些通用的模块；
- core 包：大部分核心实现代码都在本包下。其它包的代码封装上层接口，最终调用本包内代码；
- events 包：支持 event 框架；
- examples 包：包括一些示例的 chaincode 代码；
- gossip 包：实现 gossip 协议；
- msp 包：Member Service Provider 包；

- order 包：order 服务相关的入口和框架代码；
- peer 包：peer 的入口和框架代码；
- protos 包：包括各种协议和消息的 protobuf 定义文件和生成的 go 文件。

### 源码相关工具

一些辅助代码包，包括：

- bddtests：测试包，含有大量 bdd 测试用例；
- gotools：golang 开发相关工具安装；
- vendor 包：管理依赖；

### 安装部署

包括：

- busybox：busybox 环境，精简的 linux；
- devenv：配置开发环境；
- images：镜像生成模板等。
- scripts：各种安装配置脚本；

### 其它工具

其他工具，包括：

- docs：文档，大部分文档都可以 在线查阅；

## 配置、脚本和文档

除了些目录外，还包括一些说明文档、安装需求说明、License 信息文件等。

### Docker 相关文件

- .baseimage-release：生成 baseimage 时候的版本号。
- .dockerignore：生成 Docker 镜像时忽略一些目录，包括 .git 目录。

### git 相关文件

- .gitattributes：git 代码管理时候的属性文件，带有不同类型文件中换行符的规则，默认都为 linux 格式，即 `\n` 。
- .gitignore：git 代码管理时候忽略的文件和目录，包括 build 和 bin 等中间生成路径。

* .gitreview：使用 git review 时候的配置，带有项目的仓库地址信息。
* README.md：项目的说明文件，包括一些有用的链接等。

## travis 相关文件

* .travis.yml：travis 配置文件，目前是使用 golang 1.6 编辑，运行了三种测试：unit-test、behave、node-sdk-unit-tests。

## 其它

* LICENSE：Apache 2 许可文件。
* docker-env.mk：被 Makefile 引用，生成 Docker 镜像时的环节变量。
* Makefile：执行测试、格式检查、安装依赖、生成镜像等操作。
* mkdocs.yml：生成 http://hyperledger-fabric.readthedocs.io 在线文档的配置文件。
* TravisCI_Readme.md

# 核心过程

总结一些核心的过程。

# Chaincode 启动过程

## 简介

这里讲的 Chaincode 是用户链码（User Chaincode，UCC），对应用开发者来说十分重要，它提供了基于区块链分布式账本的状态处理逻辑，基于它可以开发出多种复杂的应用。

Hyperledger Fabric 中，Chaincode 默认运行在 Docker 容器中。Peer 通过调用 Docker API 来创建和启动 Chaincode 容器。Chaincode 容器启动后跟 Peer 之间创建 gRPC 连接，双方通过发送 ChaincodeMessage 来进行交互通信。Chaincode 容器利用 core.chaincode.shim 包提供的接口来向 Peer 发起请求。

## 典型结构

下面给出了链码的典型结构，用户只需要关注到 Init() 和 Invoke() 函数的实现上，在其中利用 shim.ChaincodeStubInterface 结构，实现跟账本的交互逻辑。

```
package main

import (
    "errors"
    "fmt"
    "github.com/hyperledger/fabric/core/chaincode/shim"
)

type DemoChaincode struct { }

func (t *DemoChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
    // more logics using stub here
    return stub.Success(nil)
}

func (t *DemoChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response
    // more logics using stub here
    return stub.Success(nil)
}

func main() {
    err := shim.Start(new(DemoChaincode))
    if err != nil {
        fmt.Printf("Error starting DemoChaincode: %s", err)
    }
}
```

## 启动过程

Chaincode 首先是一个普通的 Golang 程序，其 main 方法中调用了 shim 层的 Start() 方法。整体启动过程如下图所示。

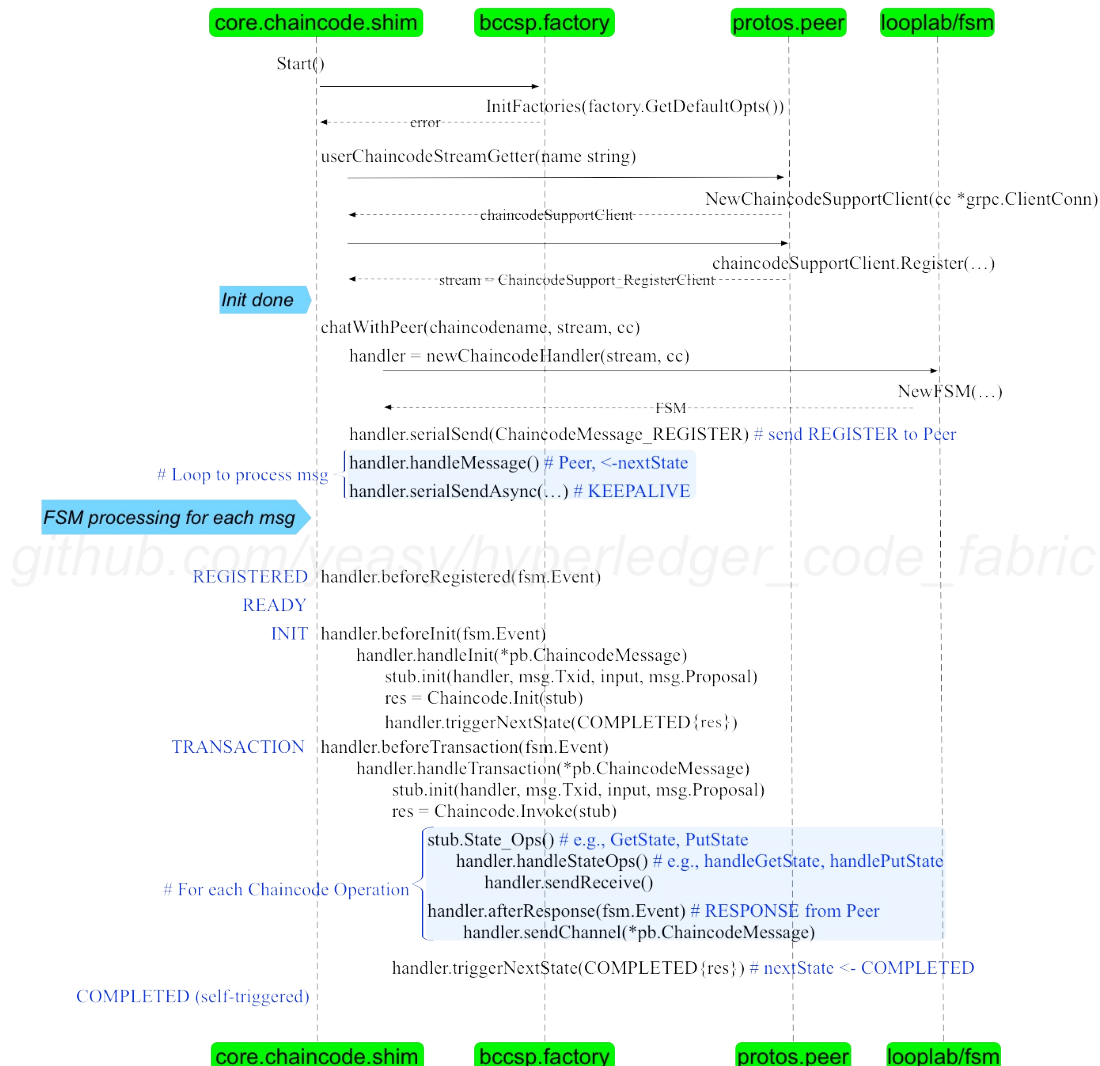



图 *1.5.1.1 - Chaincode* 启动过程

首先会进行初始化。包括读取默认配置，创建到 Peer 的gRPC 连接，主要包括 `NewChaincodeSupportClient(cc *grpc.ClientConn)` 和 `chaincodeSupportClient.Register(ctx context.Context, opts ...grpc.CallOption)` 两个方法。

初始化完成后，创建有限状态机结构（FSM，github.com/looplab/fsm）。FSM 会根据收到的消息和当前状态来触发状态转移，并执行提前设置的操作。

*Peer* 侧也利用了类似的 *FSM* 结构来管理消息响应。

之后，利用创建好的 gRPC 连接开始向 Peer 发送第一个 gRPC 消息：ChaincodeMessage_REGISTER，将自身注册到 Peer 上。注册成功后开始消息处理循环，等待接收来自 Peer 的消息以及自身的状态迁移（nextState）消息。

后续过程中，Chaincode 和 Peer 利用 FSM 完成一系列对消息的响应运作，如下所示。

- Peer 收到来自链码容器的 ChaincodeMessage_REGISTER 消息，将其注册到本地的一个 Handler 结构，返回 ChaincodeMessage_REGISTERED 消息发给链码容器。之后更新状态为 established ，并发送 ChaincodeMessage_READY 消息给链码侧，更新状态为 ready。
- 链码侧收到 ChaincodeMessage_REGISTERED 消息后，不进行任何操作，注册成功。更新状态为 established。收到 ChaincodeMessage_READY 消息后更新状态为 ready。
- Peer 侧发出 ChaincodeMessage_INIT 消息给链码容器，准备触发链码侧初始化操作。
- 链码容器收到 ChaincodeMessage_INIT 消息，通过 Handler.handleInit() 方法进行进行初始化。主要包括初始化所需的 ChaincodeStub 结构，以及调用链码代码中的 Init() 方法。初始化成功后，返回 ChaincodeMessage_COMPLETED 消息给 Peer。此时，链码容器进入可被调用（Invoke）状态。
- 链码被调用时，Peer 发出 ChaincodeMessage_TRANSACTION 消息给链码。
- 链码收到 ChaincodeMessage_TRANSACTION 消息，会调用 Invoke() 方法，根据 Invoke 方法中用户实现的逻辑，可以发出包括 ChaincodeMessage_GET_HISTORY_FOR_KEY、ChaincodeMessage_GET_QUERY_RESULT、ChaincodeMessage_GET_STATE、ChaincodeMessage_GET_STATE_BY_RANGE、ChaincodeMessage_QUERY_STATE_CLOSE、ChaincodeMessage_QUERY_STATE_NEXT、ChaincodeMessage_INVOKE_CHAINCODE 等消息给 Peer 侧。Peer 侧收到这些消息，进行相应的处理，并回复 ChaincodeMessage_RESPONSE 消息。最后，链码侧会回复调用完成的消息 ChaincodeMessage_COMPLETE 给 Peer 侧。
- 在上述过程中，Peer 和链码侧还会定期的发送 ChaincodeMessage_KEEPALIVE 消息给对方，以确保彼此在线。

# Peer 节点启动

# Peer 背书提案过程

客户端将交易预提案（Transaction Proposal)通过 gRPC 发送给支持 Endorser 角色的 Peer 进行背书。

这些交易提案可能包括链码的安装、实例化、升级、调用、查询；以及 Peer 节点加入和列出通道操作。

Peer 接收到请求后，会调用 `core/endorser/endorser.go` 中 `Endorser` 结构体的 `ProcessProposal(ctx context.Context, signedProp *pb.SignedProposal) (*pb.ProposalResponse, error)` 方法，进行具体的背书处理。

背书过程主要完成如下操作：

- 检查提案消息的合法性，以及相关的权限；
- 模拟执行提案：启动链码容器，对世界状态的最新版本进行临时快照，基于它执行链码，将结果记录在读写集中；
- 对提案内容和读写集合进行签名，并返回提案响应消息。

## 整体过程

主要过程如下图所示。

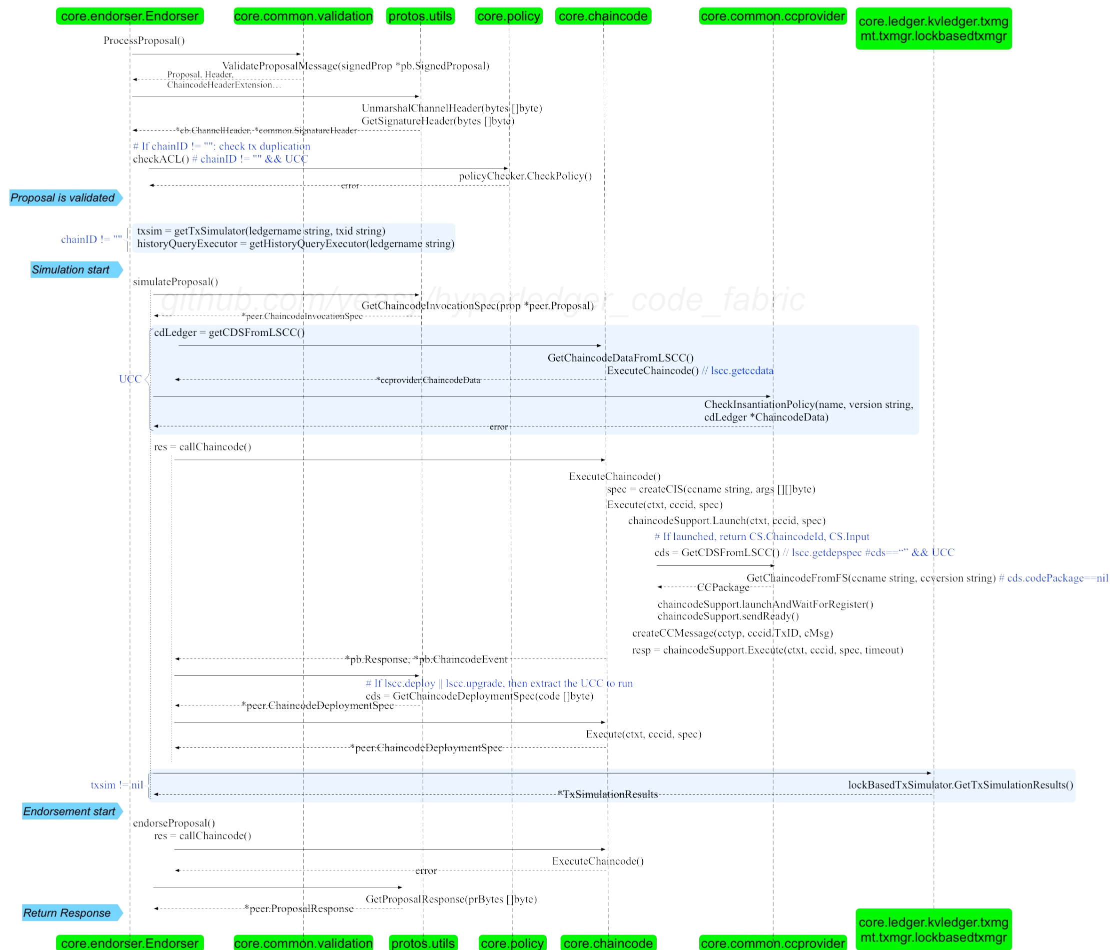



图 *1.5.3.1 - Endorser ProcessProposal* 过程

- 检查提案合法性；
  - 调用 ValidateProposalMessage() 方法对签名的提案进行格式检查，主要包括：
    - Channel 头部格式：是否合法头部类型，由 validateChannelHeader() 完成；
    - 签名头格式：是否包括了 nonce 和creators 数据，由 validateSignatureHeader() 完成；
    - 签名域：creator 证书 MSP 检查是否合法，签名是否正确，由 checkSignatureFromCreator() 完成。
  - 如果是系统链码调用（SCC），检查是否是允许从外部调用的三种 SCC 之一：cscc、lscc、qscc 或 rscc；
  - 如果 chainID 不为空，获取对应 chain 的账本结构，并检查 TxID 唯一性，确保同一交易未曾提交到账本结构中；
  - 对于用户链码调用，需要检查 ACL：资源为 `PROPOSE` ，默认策略是签名提案者在通道上拥有写权限（ `CHANNELWRITERS` ）。
- 模拟执行提案
  - 如果 chainID 不为空，获取对应账本的交易模拟器（TxSimulator）和历史查询器

（HistoryQueryExecutor），这两个结构将在后续执行链码时被使用。
    - 如果 chainID 不为空，调用 `simulateProposal()` 方法获取模拟执行的结果，检查返回的响应 response 的状态，若不小于错误 500 则创建并返回一个失败的 ProposalResponse。
- 对提案内容和读写集合进行签名
    - chainID 非空情况下，调用 endorseProposal() 方法利用 ESCC，对之前得到的模拟执行的结果进行背书。返回 ProposalResponse，检查 simulateProposal 返回的 response 的状态，若不小于错误阈值 400（被背书节点反对），返回 ProposalResponse 及链码错误 chaincodeError（endorseProposal 里有检查链码执行结果的状态，而 simulateProposal 没有检查）。
    - 将 response.Payload 赋给 ProposalResponse.Response.Payload（因为 simulateProposal 返回的 response 里面包含链码调用的结果）。
    - 返回响应消息 ProposalResponse。

## simulateProposal 方法

simulateProposal 方法会通过执行链码逻辑来获取对状态的修改结果，并存放到读写集合中，主要过程如下：

- 从提案结构的载荷中提取 ChaincodeInvocationSpec 结构，其中包含了所调用链码（包括系统链码和用户链码）的路径、名称和版本，以及调用时传入的参数列表；
- 检查 ESCC 和 VSCC（尚未实现）；
- 对用户链码，检查提案中的实例化策略跟账本上记录的该链码的实例化策略（安装链码时指定）是否一致。防止有人修改权限在其它通道非法实例化。
- 调用 callChaincode() 方法执行 Proposal，返回 Response 和 ChaincodeEvent。
    - 调用 `core.endorser` 包中 `SupportImpl.Execute()` 方法，该方法主要调用 `core.chaincode` 包中的 `ExecuteChaincode()` 方法，进一步调用包内的 `Execute()` 方法。调用过程中会把交易模拟器和历史查询器通过上下文结构体传入后续子方法。
    - `Execute()` 方法会调用 `ChaincodeSupport.Launch()` 方法创建并启动链码容器。启动成功后创建链码 gRPC 消息，通过 `ChaincodeSupport.Execute()` 方法发送消息给 CC 容器，执行相关的合约，并返回执行响应（ChaincodeMessage 结构）。此过程中会将读写集记录到交易模拟器结构体中。
- 对于非空 chainID（大部分跟账本相关的操作），执行 GetTxSimulationResults() 拿到执行结果 `TxSimulationResults` 结构，从中可以解析出读写集数据。
- 最终返回链码标准数据结构 ChaincodeDefinition、响应消息 ChaincodeMessage、交易读写集 PubSimulationResults、链码事件 ChaincodeEvent。

## endorseProposal 方法

主要过程如下：

- 获取被调用的链码指定的背书链码的名字。
- 通过 callChaincode() 实现对背书链码的调用，返回响应 response（对 ESCC 的调用同样也会产生 simulation results，但 ESCC 不能背书自己产生的simulation results，需要背书最初被调用的链码产生的 simulation results）。
- 检查 response.Status，是否大于等于 400（错误阈值），若是则把 response 赋给 proposalResponse.Response 并返回 proposalResponse。
- 将 response.Payload解码后（ProposalResponse类型）返回。

## callChaincode 方法

主要过程如下：

- 判断交易模拟器，不为空则把它加入到Context的K-V存储中。
- 判断被call的cc是不是系统链码，创建CCContext（包含通道名、链码名、版本号、交易ID、是否 SCC、签名 Prop、Prop）
- 调用 core/chaincode/chaincodeexec.go 下的 ExecuteChaincode()，返回响应 response 和 事件ccevent。
- 返回 response和ccevent。

# 排序服务核心原理和工作过程

排序服务在超级账本 Fabric 网络中起到十分核心的作用。所有交易在发送给 Committer 进行验证接受之前，需要先经过排序服务进行全局排序。

在目前架构中，排序服务的功能被抽取出来，作为单独的 fabric-orderer 模块来实现，代码主要在 `fabric/orderer` 目录下。

下面以 Kafka 作为共识插件为例，讲解 Orderer 节点的核心过程。

## 工作原理

Orderer 节点（Ordering Service Node，OSN）在网络中起到代理作用，多个 Orderer 节点会连接到 Kafka 集群，利用 Kafka 的共识功能，完成对网络中交易的排序和打包成区块的工作。

Fabric 网络提供了多通道特性，为了支持这一特性，同时保障每个 Orderer 节点上数据的一致性，排序服务进行了一些特殊设计。

对于每个通道，Orderer 将其映射到 Kafka 集群中的一个 topic （topic 名称与 channelID 相同）上。由于 Orderer 目前并没有使用 Kafka Topic 的多分区负载均衡特性，默认每个 topic 只创建了一个分区（0 号分区）。

此外，Orderer 还在本地维护了针对每个通道的账本（区块链）结构，其中每个区块包括了一组排序后的交易消息，并且被分割为独立区块。

## 核心过程

核心过程如下所示。

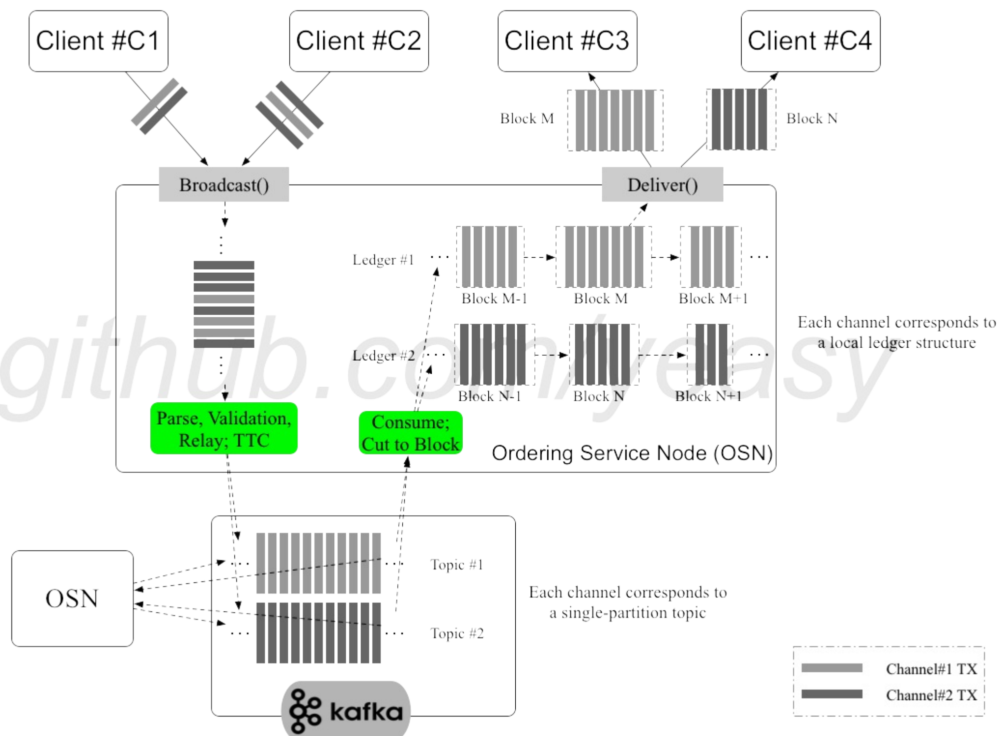



图 *1.5.5.1 - Orderer* 节点核心过程

- 客户端通过 gRPC 连接发送交易信息到 Orderer 节点的 `Broadcast()` 接口。
- Orderer 节点收到请求后，提取消息进行解析、检查，通过检查后封装为 Kafka 消息，通过 Produce 接口发送到 Kakfa 集群对应的 topic 分区中。
- 当前收到消息数达到 BatchSize.MaxMessageCount 或消息尺寸过大，或超时时间达到 BatchTimeout，则发送分块消息 TTC-X 到 Kafka。
- Kafka 集群维护多个 topic 分区。Kakfa 通过共识算法来确保写入到分区后的消息的一致性。即一旦写入分区，任何 Orderer 节点看到的都是相同的消息队列。
- Orderer 节点在启动后，还默认对本地账本对应的 Kafka 分区数据进行监听，不断从 Kafka 拉取（Consume）新的交易消息，并对消息进行处理。满足一定策略情况下（收到 TTX-C 或配置消息）还会将消息打包为区块。

## 分块决策

收到分块消息 TTC-X，或收到配置交易，则切分之前从 Kafka 中收到的消息为区块，记录到本地账本结构中。

# Orderer 节点启动过程

Orderer 节点启动通过 `orderer` 包下的 main() 方法实现，会进一步调用到 `orderer/common/server` 包中的 `Main()` 方法。

核心代码如下所示。

```
// Main is the entry point of orderer process
func Main() {
    fullCmd := kingpin.MustParse(app.Parse(os.Args[1:]))

    // "version" command
    if fullCmd == version.FullCommand() {
        fmt.Println(metadata.GetVersionInfo())
        return
    }

    conf := config.Load()
    initializeLoggingLevel(conf)
    initializeLocalMsp(conf)

    Start(fullCmd, conf)
}
```

包括配置初始化过程和核心启动过程两个部分：

- config.Load()：从本地配置文件和环境变量中读取配置信息，构建配置树结构。
- initializeLoggingLevel(conf)：配置日志级别。
- initializeLocalMsp(conf)：配置 MSP 结构。
- Start()：完成启动后的核心工作。

## 整体过程

核心启动过程都在 `orderer/common/server` 包中的 Start() 方法，如下图所示。

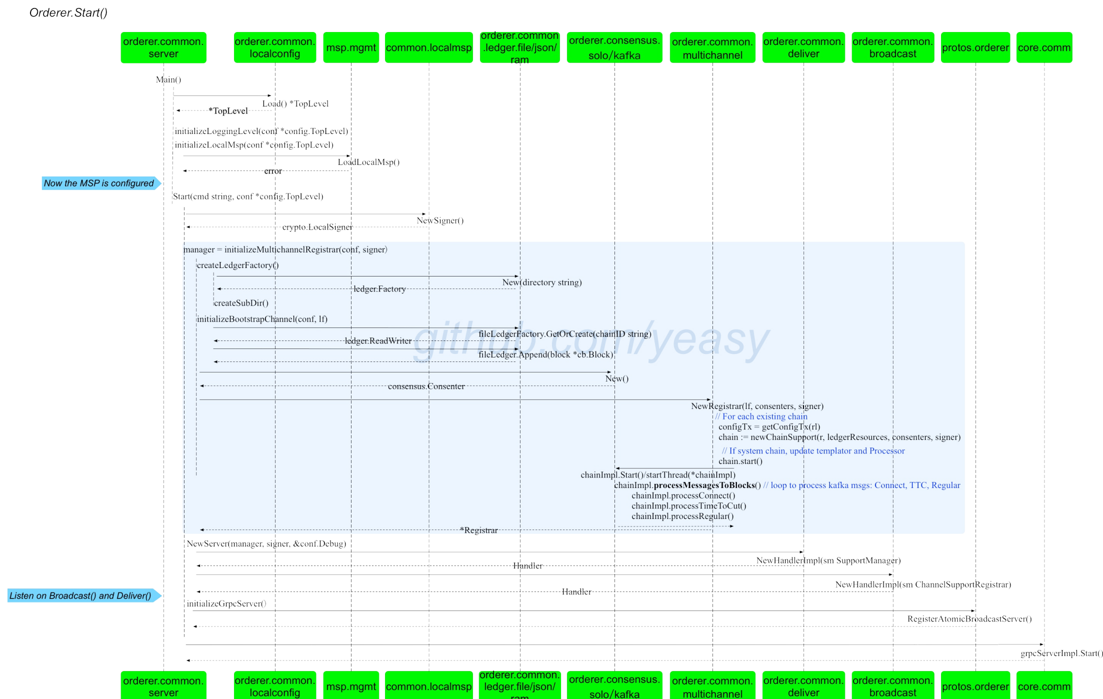



图 *1.5.6.1 - Orderer* 启动的整体过程

Start() 方法会初始化 gRPC 服务需要的结构，然后启动服务。

核心代码如下所示。

```go
func Start(cmd string, conf *config.TopLevel) {
    logger.Debugf("Start()")
    signer := localmsp.NewSigner()
    manager := initializeMultichannelRegistrar(conf, signer)
    server := NewServer(manager, signer, &conf.Debug)

    switch cmd {
    case start.FullCommand(): // "start" command
        logger.Infof("Starting %s", metadata.GetVersionInfo())
        initializeProfilingService(conf)
        grpcServer := initializeGrpcServer(conf)
        ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)
        logger.Info("Beginning to serve requests")
        grpcServer.Start()
    case benchmark.FullCommand(): // "benchmark" command
        logger.Info("Starting orderer in benchmark mode")
        benchmarkServer := performance.GetBenchmarkServer()
        benchmarkServer.RegisterService(server)
        benchmarkServer.Start()
    }
}
```

包括两大部分：

- gRPC 服务结构初始化；
- gRPC 服务启动。

## gRPC 服务结构初始化

包括创建新的 MSP 签名结构，初始化 Registrar 结构来管理各个账本结构，启动共识过程，以及创建 gRPC 服务端结构。

核心步骤包括：

```
signer := localmsp.NewSigner() // 初始化签名结构
manager := initializeMultichannelRegistrar(conf, signer, tlsCallback) // 初始化账本管理器（Registrar）结构
```

其中，`initializeMultichannelRegistrar(conf, signer)` 方法最为关键，核心代码如下：

```
func initializeMultichannelRegistrar(conf *config.TopLevel, signer crypto.LocalSigner,
 callbacks ...func(bundle *channelconfig.Bundle)) *multichannel.Registrar {
    // 创建操作账本的工厂结构
    lf, _ := createLedgerFactory(conf)

    // 如果是首次启动情况，默认先创建系统通道的本地账本结构
    if len(lf.ChainIDs()) == 0 {
        logger.Debugf("There is no chain, hence we must be in bootstrapping")
        initializeBootstrapChannel(conf, lf)
    } else {
        logger.Info("Not bootstrapping because of existing chains")
    }
    //初始化共识插件，共识插件负责跟后台的队列打交道
    consenters := make(map[string]consensus.Consenter)
    consenters["solo"] = solo.New()
    consenters["kafka"] = kafka.New(conf.Kafka.TLS, conf.Kafka.Retry, conf.Kafka.Versi
on, conf.Kafka.Verbose)

    // 创建各个账本的管理器（Registrar）结构，并启动共识过程
    return multichannel.NewRegistrar(lf, consenters, signer, callbacks...)
}
```

利用传入的配置信息和签名信息完成如下步骤：

- 创建账本操作的工厂结构；
- 如果是新启动情况，利用给定的系统初始区块文件初始化系统通道的相关结构；
- 完成共识插件（包括 `solo` 和 `kafka` 两种）的初始化；
- `multichannel.NewRegistrar(lf, consenters, signer)` 方法会扫描本地账本数据（此时至少已存在系统通道），创建 Registrar 结构，并为每个账本都启动共识（如 Kafka 排序）

过程。

说明：*Registrar* 结构（位于 `orderer.common.multichannel` 包）是 *Orderer* 组件中最核心的结构，管理了 *Orderer* 中所有的账本、共识插件等数据结构。

## 创建 Registrar 结构并启动共识过程

`NewRegistrar(lf, consenters, signer)` 方法位于 `orderer.common.multichannel` 包，负责初始化链支持、消息处理器等重要数据结构，并为各个账本启动共识过程。

核心代码如下：

```
existingChains := ledgerFactory.ChainIDs()
for _, chainID := range existingChains { // 启动本地所有的账本结构的共识过程
    if _, ok := ledgerResources.ConsortiumsConfig(); ok { // 如果是系统账本（默认在首次启动时会自动创建）
        chain := newChainSupport(r, ledgerResources, consenters, signer)
        chain.Processor = msgprocessor.NewSystemChannel(chain, r.templator, msgprocessor.CreateSystemChannelFilters(r, chain))
        r.chains[chainID] = chain
        r.systemChannelID = chainID
        r.systemChannel = chain
        defer chain.start() // 启动共识过程
    else // 如果是应用账本
        chain := newChainSupport(r, ledgerResources, consenters, signer)
        r.chains[chainID] = chain
        chain.start()  // 启动共识过程
    }
```

`chain.start()` 方法负责启动共识过程。以 Kafka 共识插件为例，最终以协程方式调用到 `orderer.consensus.kafka` 包中的 `startThread()` 方法，将在后台持续运行。

```
func (chain *chainImpl) Start() {
    go startThread(chain)
}
```

`startThread()` 方法将为指定的账本结构配置共识服务，并将其启动，核心代码包括：

```
// 创建 Producer 结构
chain.producer, err = setupProducerForChannel(chain.consenter.retryOptions(), chain.haltChan, chain.SharedConfig().KafkaBrokers(), chain.consenter.brokerConfig(), chain.channel)
// 发送 CONNECT 消息给 Kafka，如果失败，则退出
sendConnectMessage(chain.consenter.retryOptions(), chain.haltChan, chain.producer, chain.channel)

// 创建处理对应 Kafka topic 的 Consumer 结构
chain.parentConsumer, err = setupParentConsumerForChannel(chain.consenter.retryOptions(), chain.haltChan, chain.SharedConfig().KafkaBrokers(), chain.consenter.brokerConfig(), chain.channel)
// 配置从指定 partition 读取消息的 PartitionConsumer 结构
chain.channelConsumer, err = setupChannelConsumerForChannel(chain.consenter.retryOptions(), chain.haltChan, chain.parentConsumer, chain.channel, chain.lastOffsetPersisted+1)

// 从该链对应的 Kafka 分区不断读取消息，并进行处理过程
chain.processMessagesToBlocks()
```

主要包括如下步骤：

- 创建到 Kafka 集群的 Producer 结构并发送 CONNECT 消息；
- 为对应的 topic 创建 Consumer 结构，并配置从指定分区读取消息的 PartitionConsumer 结构；
- 对链对应的 Kafka 分区中消息的进行循环处理。这部分更详细内容可以参考 Orderer 节点对排序后消息的处理过程。

## gRPC 服务启动

初始化 gRPC 服务结构，完成绑定并启动监听。

```
// 初始化 gRPC 服务端结构
server := NewServer(manager, signer, &conf.Debug)

// 创建 gRPC 服务连接
grpcServer := initializeGrpcServer(conf)

// 绑定 gRPC 服务并启动
ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)
grpcServer.Start()
```

其中，`NewServer(manager, signer, &conf.Debug)` 方法（位于 `orderer.common.server` 包）最为核心，将 gRPC 相关的服务结构进行初始化，并绑定到 gRPC 请求上。分别响应 Deliver() 和 Broadcast() 两个 gRPC 调用。

```
// NewServer creates an ab.AtomicBroadcastServer based on the broadcast target and led
ger Reader
func NewServer(r *multichannel.Registrar, _ crypto.LocalSigner, debug *localconfig.Deb
ug) ab.AtomicBroadcastServer {
    s := &server{
        dh:    deliver.NewHandlerImpl(deliverSupport{Registrar: r}),
        bh:    broadcast.NewHandlerImpl(broadcastSupport{Registrar: r}),
        debug: debug,
    }
    return s
}
```

# Orderer 节点对排序后消息的处理过程

经过排序后的消息，可以认为在网络中已经达成了基本的共识。Orderer 会获取这些消息，进行对应处理（包括打包为区块，更新本地账本结构等）。

以 Kafka 模式为例，Orderer 节点启动后，会调用 `orderer/consensus/kafka` 模块中 `chainImpl` 结构体的 `processMessagesToBlocks() ([]uint64, error)` 方法，持续获取 Kafka 对应分区中的消息。

## 主要过程

`chainImpl` 结构体的 `processMessagesToBlocks()` 方法不断从分区中 Consume 消息并进行处理，同时定时发送 TimeToCut 消息。

处理消息类型包括 Connect 消息（Producer 启动后发出）、TimeToCut 消息和 Regular 消息（Fabric 消息）。分别调用对应方法进行处理，主要流程如下：

```
// orderer/consensus/kafka/chain.go
for {
    select {
        case <-chain.haltChan: // 链故障了，退出
        case kafkaErr := <-chain.channelConsumer.Errors(): //获取 Kakfa 消息发生错误
            select {
                case <-chain.errorChan: // 连接关闭，不进行任何操作
                default: //其它错误，OutofRange，关闭 errorChan；否则进行超时重连
            }
            select {
                case <-chain.errorChan: // 连接仍然关闭，尝试后台进行重连
            }
        case <-topicPartitionSubscriptionResumed: // 继续
        case <-deliverSessionTimedOut: //访问超时，尝试后台进行重连
        case in, ok := <-chain.channelConsumer.Messages(): // 核心过程：成功读取到 Kafka
消息，进行处理
            switch msg.Type.(type) {
            case *ab.KafkaMessage_Connect: // Kafka 连接消息，忽略
            case *ab.KafkaMessage_TimeToCut: // TTC，打包现有的一批消息为区块
            case *ab.KafkaMessage_Regular: // 核心处理：Fabric 相关消息，包括配置更新、应用通
道交易等
                chain.processRegular(msg.GetRegular(), in.Offset)
        case <-chain.timer: //定期发出 TimeToCut 消息到 Kafka
    }
}
```

## Fabric 相关消息的处理

对于 Fabric 相关消息（包括交易消息和配置消息），具体会调用 chainImpl 结构体的 `processRegular(regularMessage *ab.KafkaMessageRegular, receivedOffset int64) error` 方法进行处理。

该方法的核心代码如下：

```
// orderer/consensus/kafka/chain.go
func (chain *chainImpl) processRegular(regularMessage *ab.KafkaMessageRegular, receive
dOffset int64) error {
    env := &cb.Envelope{}
    proto.Unmarshal(regularMessage.Payload, env) // 从载荷中解析出信封结构
    switch regularMessage.Class {
        case ab.KafkaMessageRegular_NORMAL: // 普通交易消息
            chain.ProcessNormalMsg(env) // 检查消息合法性，分应用链和系统链两种情况
            commitNormalMsg(env) // 处理交易消息，满足条件则切块，并写入本地账本
        case ab.KafkaMessageRegular_CONFIG: // 配置消息，包括通道头部类型为 CONFIG、CONFIG_
UPDATE、ORDERER_TRANSACTION 三种
            chain.ProcessConfigMsg(env) //检查消息合法性，分应用链和系统链两种情况
            commitConfigMsg(env) // 切块，写入账本。如果是 ORDERER_TRANSACTION 消息，创建新
的应用通道账本；如果是 CONFIG 消息，更新配置。
}
```

## 普通交易消息

普通交易消息，会检查是否满足生成区块的条件，满足则产生区块并写入本地账本结构。通过内部的 `commitNormalMsg(env)` 方法来完成。

该方法主要调用 `orderer/common/multichannel` 模块中 `BlockWriter` 结构体的 `CreateNextBlock(messages []*cb.Envelope) *cb.Block` 方法和 `WriteBlock(block *cb.Block, encodedMetadataValue []byte)` 方法。

`CreateNextBlock(messages []*cb.Envelope) *cb.Block` 方法基本过程十分简单，创建新的区块，将传入的交易的信封结构直接序列化到 `block.Data.Data[]` 域中。

```
// orderer/common/multichannel/blockwriter.go
func (bw *BlockWriter) CreateNextBlock(messages []*cb.Envelope) *cb.Block {
    previousBlockHash := bw.lastBlock.Header.Hash()

    data := &cb.BlockData{
        Data: make([][]byte, len(messages)),
    }

    var err error
    for i, msg := range messages {
        data.Data[i], err = proto.Marshal(msg)
        if err != nil {
            logger.Panicf("Could not marshal envelope: %s", err)
        }
    }

    block := cb.NewBlock(bw.lastBlock.Header.Number+1, previousBlockHash)
    block.Header.DataHash = data.Hash()
    block.Data = data

    return block
}
```

`WriteBlock(block *cb.Block, encodedMetadataValue []byte)` 方法则将 Kafka 相关的元数据也附加到区块结构中，添加区块的签名、最新配置的签名，并写入到本地账本。

```
// orderer/common/multichannel/blockwriter.go
func (bw *BlockWriter) WriteBlock(block *cb.Block, encodedMetadataValue []byte) {
    bw.committingBlock.Lock()
    bw.lastBlock = block

    go func() {
        defer bw.committingBlock.Unlock()
        bw.commitBlock(encodedMetadataValue)
    }()
}
```

Kafka 相关的元数据（KafkaMetadata）包括：

- LastOffsetPersisted：上次消息的偏移量；
- LastOriginalOffsetProcessed：本条消息被重复处理时，最新的偏移量；
- LastResubmittedConfigOffset：上次提交的配置消息的偏移量。

## 配置交易消息

首先会检查消息中配置版本号是否跟当前链上的配置版本号一致。

如果不一致，则会更新后生成新的配置信封消息，扔回到后端的共识模块（如 Kafka），并且阻塞新的 Broadcast 消息直到重新提交的消息得到处理。代码片段如下：

```
// orderer/consensus/kafka/chain.go
if regularMessage.ConfigSeq < seq { // 消息中配置版本并非最新版本
    configEnv, configSeq, err := chain.ProcessConfigMsg(env)

    // For both messages that are ordered for the first time or re-ordered, we set original offset
    // to current received offset and re-order it.
    if err := chain.configure(configEnv, configSeq, receivedOffset); err != nil {
        return fmt.Errorf("error re-submitting config message because = %s", err)
    }

    chain.lastResubmittedConfigOffset = receivedOffset      // Keep track of last resubmitted message offset
    chain.doneReprocessingMsgInFlight = make(chan struct{}) // Create the channel to block ingress messages

    return nil
}
```

如果版本一致，则调用内部的 `commitConfigMsg(env)` 方法根据信封结构来产生区块。

```
commitConfigMsg := func(message *cb.Envelope, newOffset int64) {
    batch := chain.BlockCutter().Cut() // 尝试把收到的交易汇总

    if batch != nil { // 如果已经积累了一些交易，则先把它们打包为区块
        block := chain.CreateNextBlock(batch)
        metadata := utils.MarshalOrPanic(&ab.KafkaMetadata{
            LastOffsetPersisted:         receivedOffset - 1,
            LastOriginalOffsetProcessed: chain.lastOriginalOffsetProcessed,
            LastResubmittedConfigOffset: chain.lastResubmittedConfigOffset,
        })
        chain.WriteBlock(block, metadata)
        chain.lastCutBlockNumber++
    }

    chain.lastOriginalOffsetProcessed = newOffset
    block := chain.CreateNextBlock([]*cb.Envelope{message}) // 将配置交易生成区块
    metadata := utils.MarshalOrPanic(&ab.KafkaMetadata{
        LastOffsetPersisted:         receivedOffset,
        LastOriginalOffsetProcessed: chain.lastOriginalOffsetProcessed,
        LastResubmittedConfigOffset: chain.lastResubmittedConfigOffset,
    })
    chain.WriteConfigBlock(block, metadata) // 添加区块到系统链
    chain.lastCutBlockNumber++
    chain.timer = nil
}
```

由于每个配置消息会单独生成区块。因此，如果之前已经收到了一些普通交易消息，会先把这些消息生成区块。

接下来，调用 `orderer/common/multichannel` 模块中 `BlockWriter` 结构体的 `CreateNextBlock(messages []*cb.Envelope) *cb.Block` 方法和 `WriteConfigBlock(block *cb.Block, encodedMetadataValue []byte)` 方法来分别打包区块和更新账本结构，代码如下：

```
// orderer/consensus/kafka/chain.go
chain.lastOriginalOffsetProcessed = newOffset
block := chain.CreateNextBlock([]*cb.Envelope{message})
metadata := utils.MarshalOrPanic(&ab.KafkaMetadata{
    LastOffsetPersisted:         receivedOffset,
    LastOriginalOffsetProcessed: chain.lastOriginalOffsetProcessed,
    LastResubmittedConfigOffset: chain.lastResubmittedConfigOffset,
})
chain.WriteConfigBlock(block, metadata)
chain.lastCutBlockNumber++
chain.timer = nil
```

其中，`WriteConfigBlock()` 方法执行解析消息和处理的主要逻辑，核心代码如下所示。

```
// orderer/common/multichannel/blockwriter.go
func (bw *BlockWriter) WriteConfigBlock(block *cb.Block, encodedMetadataValue []byte)
{

    // 解析配置交易信封结构，每个区块中只有一个配置交易
    ctx, err := utils.ExtractEnvelope(block, 0)

    // 解析载荷和通道头结构
    payload, err := utils.UnmarshalPayload(ctx.Payload)
    chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)

    // 按照配置交易内容，执行对应操作
    switch chdr.Type { // 排序后只有 ORDERER_TRANSACTION 和 CONFIG 两种类型消息
        case int32(cb.HeaderType_ORDERER_TRANSACTION): // 新建应用通道
            newChannelConfig, err := utils.UnmarshalEnvelope(payload.Data)

            // 创建新的本地账本结构并启动对应的轮询消息过程，实际调用 orderer/common/multichann
el.Registrar.newChain(configtx *cb.Envelope)
            bw.registrar.newChain(newChannelConfig)
        case int32(cb.HeaderType_CONFIG): // 更新通道配置
            configEnvelope, err := configtx.UnmarshalConfigEnvelope(payload.Data)
            bundle, err := bw.support.CreateBundle(chdr.ChannelId, configEnvelope.Conf
ig)
            bw.support.Update(bundle)

    }

    // 将区块写入到本地账本结构
    bw.WriteBlock(block, encodedMetadataValue)
}
```

# Orderer 节点 Broadcast 请求的处理

Broadcast，意味着客户端将请求消息（例如完成背书后的交易）通过 gRPC 接口发送给 Ordering 服务。Orderer 进行本地验证处理后，会转化为入队消息发给后端共识模块（如 Kafka）。

发给 Orderer 的 Broadcast 请求消息包括链码的实例化、调用；通道的创建、更新。

来自客户端的请求消息，会首先交给 `orderer.common.server` 包中 server 结构体的 `Broadcast(srv ab.AtomicBroadcast_BroadcastServer) error` 方法处理。该方法主要会调用到 `orderer.common.broadcast` 包中 handlerImpl 结构的 `Handle(srv ab.AtomicBroadcast_BroadcastServer) error` 方法。

handlerImpl 结构体十分重要，在 Orderer 整个处理过程中都会用到。

```
type handlerImpl struct {
    sm ChannelSupportRegistrar
}

func (bh *handlerImpl) Handle(srv ab.AtomicBroadcast_BroadcastServer) error
```

## 整体过程

Broadcast 请求的整体处理过程如下图所示。

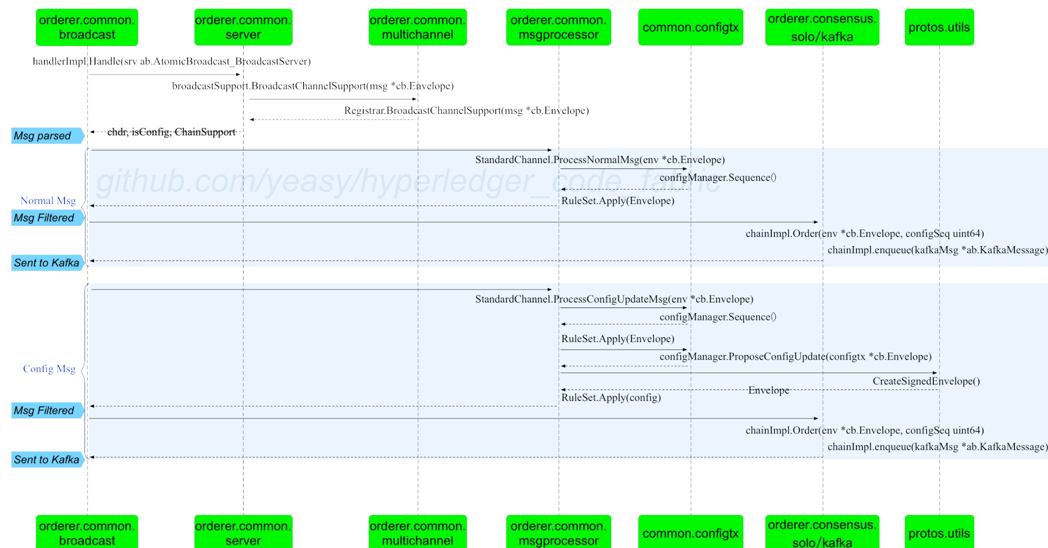



图 *1.5.8.1 - Orderer* 节点 *Broadcast* 处理过程

`Handle(srv ab.AtomicBroadcast_BroadcastServer) error` 方法会开启一个循环来从 srv 中读取请求消息并进行处理，直到结束。主要包括解析消息、处理消息（包括配置消息和非配置消息）和返回响应三个步骤。

核心代码如下所示（位于 `orderer/common/broadcast/broadcast.go#handlerImpl.Handle()`）：

```
for {
    msg, error := srv.Recv() // 从请求中提取一个 Envelope 消息

    // 解析消息：判断是否为配置消息；获取对应本地账本结构：由通道头部中指定的通道 ID 决定，本地对应账本结构不存在时（如新建应用通道）则由系统通道来处理
    chdr, isConfig, processor, err := bh.sm.BroadcastChannelSupport(msg)
    // 检查是否被之前重新提交的消息阻塞
    processor.WaitReady()

    // 对应的通道结构对消息进行处理
    if !isConfig { // 普通消息
        configSeq, err := processor.ProcessNormalMsg(msg) //消息检查
        processor.Order(msg, configSeq) //入队列操作
    } else { // 配置消息，目前只有 CONFIG_UPDATE 类型，如创建、更新通道，或获取配置区块
        config, configSeq, err := processor.ProcessConfigUpdateMsg(msg) // 合并配置更新消息
        processor.Configure(config, configSeq) //入队列操作：相关处理后发给 Kafka
    }

    srv.Send(&ab.BroadcastResponse{Status: cb.Status_SUCCESS}) // 返回响应消息
}
```

分为三个步骤：

- 解析消息：判断是否为配置消息，决定消息应由哪个通道结构进行处理，注意对于创建应用通道消息，处理器指定为系统的通道结构；
- 处理消息：选用对应的通道结构对消息进行处理，包括普通消息和配置消息；
- 返回响应消息给请求方。

下面分别进行剖析。

## 解析消息

首先，解析消息，获取消息通道头、是否为配置消息、获取对应处理器结构（链结构）。

```
chdr, isConfig, processor, err := bh.sm.BroadcastChannelSupport(msg)
```

实际上，会映射到 orderer.common.server 包中 broadcastSupport 结构体的 `BroadcastChannelSupport(msg *cb.Envelope) (*cb.ChannelHeader, bool, broadcast.ChannelSupport, error)` 方法，进一步调用到 orderer.common.multichannel 包中 Registrar 结构体的对应方法。

```
// orderer/common/multichannel/registrar.go
func (r *Registrar) BroadcastChannelSupport(msg *cb.Envelope) (*cb.ChannelHeader, bool
, *ChainSupport, error) {
    chdr, err := utils.ChannelHeader(msg)
    if err != nil {
        return nil, false, nil, fmt.Errorf("could not determine channel ID: %s", err)
    }

    cs, ok := r.chains[chdr.ChannelId] // 应用通道、系统通道
    if !ok {
        cs = r.systemChannel // 空，则默认为系统通道，如收到新建应用通道请求时，Orderer 本地并
没有该应用通道结构
    }

    isConfig := false
    switch cs.ClassifyMsg(chdr) { // 只有 CONFIG_UPDATE 会返回 ConfigUpdateMsg
    case msgprocessor.ConfigUpdateMsg: // CONFIG_UPDATE 消息，包括创建、更新通道，获取配置区
块等
        isConfig = true
    default:
    }

    return chdr, isConfig, cs, nil
}
```

channel 头部从消息信封结构中解析出来；是否为配置信息根据消息头中通道类型进行判断（是否为 cb.HeaderType_CONFIG_UPDATE）；通过字典结构查到对应的 ChainSupport 结构（应用通道、系统通道）作为处理器。

之后，利用解析后的结果，分别对不同类型的消息（普通消息、配置消息）进行不同处理。

下面默认以常见的应用通道场景进行介绍。

## 处理普通交易消息

对于普通交易消息，主要执行如下两个操作：消息格式检查和入队列操作。

```
configSeq, err := processor.ProcessNormalMsg(msg) //消息检查
processor.Order(msg, configSeq) //入队列操作
```

## 消息格式检查

消息检查方法会映射到 orderer.common.msgprocessor 包中 StandardChannel/SystemChannel 结构体的 `ProcessNormalMsg(env *cb.Envelope) (configSeq uint64, err error)` 方法，以应用通道为例，实现如下。

```
// orderer/common/msgprocessor/standardchannel.go
func (s *StandardChannel) ProcessNormalMsg(env *cb.Envelope) (configSeq uint64, err error) {
    configSeq = s.support.Sequence() // 获取配置的序列号，映射到 common.configtx 包中 configManager 结构体的对应方法
    err = s.filters.Apply(env) // 进行过滤检查，实现为 orderer.common.msgprocessor 包中 RuleSet 结构体的对应方法。
    return
}
```

其中，过滤器会在创建 ChainSupport 结构时候初始化：

- 应用通道：orderer.common.mspprocessor 包中的 `CreateStandardChannelFilters(filterSupport channelconfig.Resources) *RuleSet` 方法，包括 EmptyRejectRule、SizeFilter 和 SigFilter（ChannelWriters 角色）。
- 系统通道：orderer.common.mspprocessor 包中的 `CreateSystemChannelFilters(chainCreator ChainCreator, ledgerResources channelconfig.Resources) *RuleSet` 方法，包括 EmptyRejectRule、SizeFilter、SigFilter（ChannelWriters 角色）和 SystemChannelFilter。

## 入队列操作

入队列操作会根据 consensus 配置的不同映射到 orderer.consensus.solo 包或 orderer.consensus.kafka 包中的方法。

以 kafka 情况为例，会映射到 chainImpl 结构体的对应方法。该方法会将消息进一步封装为 `sarama.ProducerMessage` 类型消息，通过 enqueue 方法发给 Kafka 后端。

```
// orderer/consensus/kafka/chain.go#chainImpl.Order()
func (chain *chainImpl) Order(env *cb.Envelope, configSeq uint64) error {
    return chain.order(env, configSeq, int64(0))
}

func (chain *chainImpl) order(env *cb.Envelope, configSeq uint64, originalOffset int64
) error {
    marshaledEnv, err := utils.Marshal(env)
    if err != nil {
        return fmt.Errorf("cannot enqueue, unable to marshal envelope because = %s", e
rr)
    }
    if !chain.enqueue(newNormalMessage(marshaledEnv, configSeq, originalOffset)) {
        return fmt.Errorf("cannot enqueue")
    }
    return nil
}
```

## 处理配置交易消息

对于配置交易消息（CONFIG_UPDATE 类型消息，包括创建、更新通道，获取配置区块等），处理过程与正常消息略有不同，包括合并配置更新消息和入队列操作两个操作。

## 合并配置更新

主要过程包括如下两个步骤：

```
config, configSeq, err := processor.ProcessConfigUpdateMsg(msg) // 合并配置更新，生成新的配置信封结构
processor.Configure(config, configSeq) //入队列操作，将生成的配置信封结构消息扔给后端队列（如 Kafka）
```

其中，合并配置更新消息方法会映射到 `orderer.common.msgprocessor` 包中 `StandardChannel/SystemChannel` 结构体的 `ProcessConfigUpdateMsg(env *cb.Envelope) (configSeq uint64, err error)` 方法，计算合并后的配置和配置编号。

以应用通道为例，实现如下。

```
// orderer/common/msgprocessor/standardchannel.go
func (s *StandardChannel) ProcessConfigUpdateMsg(env *cb.Envelope) (config *cb.Envelop
e, configSeq uint64, err error) {
    logger.Debugf("Processing config update message for channel %s", s.support.ChainID
())

    seq := s.support.Sequence() // 获取当前配置的版本号
    err = s.filters.Apply(env) // 校验权限，是否可以更新配置
    if err != nil {
        return nil, 0, err
    }

    // 根据输入的更新配置交易消息生成配置信封结构：Config 为更新后配置字典；LastUpdate 为输入的更
新配置交易
    // 最终调用 `common/configtx` 包下 `ValidatorImpl.ProposeConfigUpdate()` 方法。
    configEnvelope, err := s.support.ProposeConfigUpdate(env)
    if err != nil {
        return nil, 0, err
    }

    // 生成签名的配置信封结构，通道头类型为 HeaderType_CONFIG。即排序后消息类型将由 CONFIG_UPDAT
E 变更为 CONFIG
    config, err = utils.CreateSignedEnvelope(cb.HeaderType_CONFIG, s.support.ChainID()
, s.support.Signer(), configEnvelope, msgVersion, epoch)
    if err != nil {
        return nil, 0, err
    }

    err = s.filters.Apply(config) // 校验生成的配置消息是否合法
    if err != nil {
        return nil, 0, err
    }

    return config, seq, nil
}
```

对于系统通道情况，除了调用普通通道结构的对应方法来处理普通的更新配置交易外，还会负责新建应用通道请求。

```
// orderer/common/msgprocessor/systemchannel.go
func (s *SystemChannel) ProcessConfigUpdateMsg(envConfigUpdate *cb.Envelope) (config *
cb.Envelope, configSeq uint64, err error) {
    channelID, err := utils.ChannelID(envConfigUpdate)
    if channelID == s.support.ChainID() { // 更新系统通道的配置交易，与普通通道相同处理
        return s.StandardChannel.ProcessConfigUpdateMsg(envConfigUpdate)
    }

    // 从系统通道中获取当前最新的配置
    // orderer/common/msgprocessor/systemchannel.go#DefaultTemplator.NewChannelConfig()

    bundle, err := s.templator.NewChannelConfig(envConfigUpdate)

    // 合并来自客户端的配置更新信封结构，创建配置信封结构 ConfigEnvelope
    newChannelConfigEnv, err := bundle.ConfigtxValidator().ProposeConfigUpdate(envConf
igUpdate)

    // 封装新的签名信封结构，其 Payload.Data 是 newChannelConfigEnv
    newChannelEnvConfig, err := utils.CreateSignedEnvelope(cb.HeaderType_CONFIG, chann
elID, s.support.Signer(), newChannelConfigEnv, msgVersion, epoch)

    // 处理新建应用通道请求，封装为 ORDERER_TRANSACTION 类型消息
    wrappedOrdererTransaction, err := utils.CreateSignedEnvelope(cb.HeaderType_ORDERER
_TRANSACTION, s.support.ChainID(), s.support.Signer(), newChannelEnvConfig, msgVersion
, epoch)

    s.StandardChannel.filters.Apply(wrappedOrdererTransaction) // 再次校验配置

    // 返回封装后的签名信封结构
    return wrappedOrdererTransaction, s.support.Sequence(), nil
}
```

## 入队列操作

入队列操作会根据 consensus 配置的不同映射到 `orderer.consensus.solo` 包或 `orderer.consensus.kafka` 包中的方法。

以 kafka 情况为例，会映射到 `chainImpl` 结构体的 `Configure(config *cb.Envelope, configSeq uint64)` 方法。该方法会调用 `configure(config *cb.Envelope, configSeq uint64, originalOffset int64)` 方法，将消息进一步封装为 `KafkaMessage_Regular` 类型消息，通过 enqueue 方法发给 Kafka 后端。

```
// orderer/consensus/kafka/chain.go
func (chain *chainImpl) configure(config *cb.Envelope, configSeq uint64, originalOffse
t int64) error {
    marshaledConfig, err := utils.Marshal(config)
    if err != nil {
        return fmt.Errorf("cannot enqueue, unable to marshal config because %s", err)
    }

    // 封装为 `KafkaMessageRegular_CONFIG` 类型消息，并通过 producer 发给 Kafka
    if !chain.enqueue(newConfigMessage(marshaledConfig, configSeq, originalOffset)) {
        return fmt.Errorf("cannot enqueue")
    }
    return nil
}
```

其中，封装为 `KafkaMessageRegular_CONFIG` 类型消息过程十分简单。

```
// orderer/consensus/kafka/chain.go
func newConfigMessage(config []byte, configSeq uint64, originalOffset int64) *ab.Kafka
Message {
    return &ab.KafkaMessage{
        Type: &ab.KafkaMessage_Regular{
            Regular: &ab.KafkaMessageRegular{
                Payload:        config,
                ConfigSeq:      configSeq,
                Class:          ab.KafkaMessageRegular_CONFIG,
                OriginalOffset: originalOffset,
            },
        },
    }
}
```

之后 Orderer 将再次从 Kakfa 获取到共识（这里主要是排序）完成的 `KafkaMessageRegular_CONFIG` 消息，进行解析和处理。具体可以参考 Orderer 节点对排序后消息的处理过程。

## 返回响应

如果处理成功，则返回成功响应消息。

```
srv.Send(&ab.BroadcastResponse{Status: cb.Status_SUCCESS})
```

# Orderer 节点 Deliver 请求的处理

Deliver，意味着客户端通过 gRPC 接口从 Ordering 服务获取数据（例如指定区块的数据）。

Orderer 节点收到请求消息，会首先交给 `orderer.common.server` 包中 server 结构体的 `Deliver(srv ab.AtomicBroadcast_DeliverServer) error` 方法处理。该方法进一步调用 `orderer.common.deliver` 包中 deliverServer 结构的 `Handle(srv ab.AtomicBroadcast_DeliverServer) error` 方法进行处理。

deliverServer 结构体十分重要，完成对 Deliver 请求的处理过程。

```
type deliverServer struct {
    sm SupportManager
}

func (ds *deliverServer) Handle(srv ab.AtomicBroadcast_DeliverServer) error
```

## 整体过程

整体处理过程如下图所示。

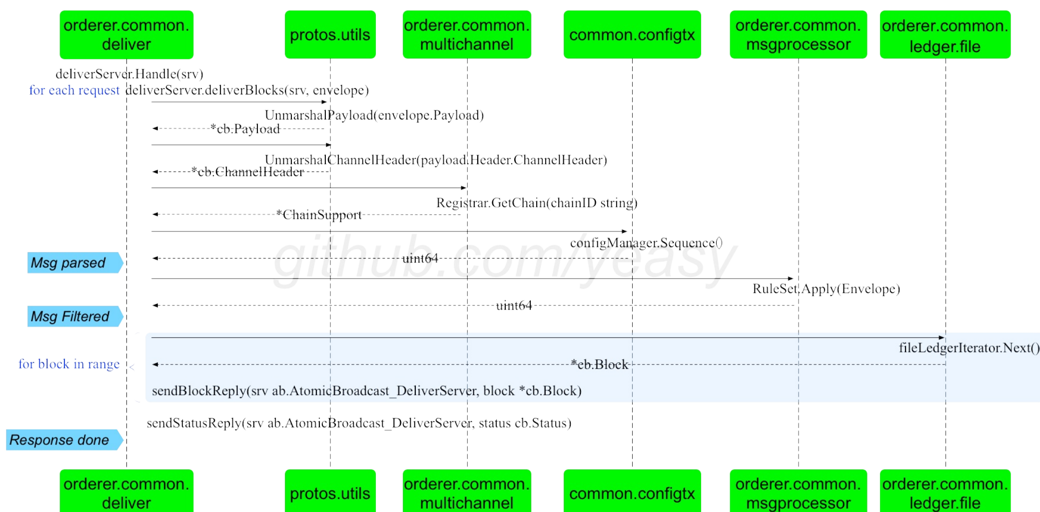



图 *1.5.9.1 - Orderer* 节点 *Deliver* 处理过程

`Handle(srv ab.AtomicBroadcast_DeliverServer) error` 方法会开启一个循环来从 srv 中不断读取请求消息并进行处理，直到结束。

核心代码如下所示，包括提取消息和对消息进行处理两个步骤。

```
for {
    envelope, err := srv.Recv() // 从请求中提取一个 Envelope 消息
    ds.deliverBlocks(srv, envelope) // 对消息进行处理并答复，核心过程
}
```

可见，对单个请求的处理都在 `deliverBlocks(srv ab.AtomicBroadcast_DeliverServer, envelope *cb.Envelope)` 方法中。该方法的处理过程包括解析消息、检查合法性、发送区块以及返回响应四个步骤。

下面具体对其进行具体分析。

## 解析消息

首先，从请求的 Envelope 结构中提取载荷（Payload），进一步从载荷中提取通道头部信息。利用通道头部信息获取对应的本地链结构，并获取当前最新的配置序列号。

```
// 提取载荷
payload, err := utils.UnmarshalPayload(envelope.Payload)

// 提取通道头
chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)

// 获取链结构，映射到 orderer.common.multichannel 包中 Registrar 结构体中对应方法
chain, ok := ds.sm.GetChain(chdr.ChannelId)

// 获取当前配置序列号
lastConfigSequence := chain.Sequence()
```

## 检查合法性

包括对权限和 seekInfo 数据进行检查。

首先，检查请求方是否对通道拥有读权限。

```
sf := msgprocessor.NewSigFilter(policies.ChannelReaders, chain.PolicyManager())
if err := sf.Apply(envelope); err != nil {
    logger.Warningf("[channel: %s] Received unauthorized deliver request from %s: %s",
 chdr.ChannelId, addr, err)
    return sendStatusReply(srv, cb.Status_FORBIDDEN)
}
```

接下来，从 Envelope 结构的 payload.data 域中解析出 seekInfo 结构，并检查其合法性。

```
proto.Unmarshal(payload.Data, seekInfo)
chain.Reader().Iterator(seekInfo.Start)

// 检查 seekInfo 的
cursor, number := chain.Reader().Iterator(seekInfo.Start)
switch stop := seekInfo.Stop.Type.(type) {
case *ab.SeekPosition_Oldest: // 截止到最早的区块
    stopNum = number
case *ab.SeekPosition_Newest: // 截止到最新的区块
    stopNum = chain.Reader().Height() - 1
case *ab.SeekPosition_Specified: // 截止到特定的区块
    stopNum = stop.Specified.Number
    if stopNum < number {
        logger.Warningf("[channel: %s] Received invalid seekInfo message from %s: star
t number %d greater than stop number %d", chdr.ChannelId, addr, number, stopNum)
        return sendStatusReply(srv, cb.Status_BAD_REQUEST)
    }
}
```

## 发送区块

在指定的起始和截止范围内，逐个从本地账本读取区块，并发送对应的区块数据，

核心代码如下所示。

```
for {
    block, status := cursor.Next() // 获取区块
    sendBlockReply(srv, block) // 发送区块
    if stopNum == block.Header.Number {
        break
    }
}
```

## 返回响应

如果处理成功，则返回成功响应消息。

```
sendStatusReply(srv, cb.Status_SUCCESS)
```

# 通道创建

主要步骤包括：

- 客户端调用 sendCreateChainTransaction()，检查指定的配置交易文件，或者利用默认配置，构造一个创建应用通道的配置交易结构，封装为 Envelope，指定 channel 头部类型为 CONFIG_UPDATE。
- 客户端发送配置交易到 Ordering 服务。
- Orderer 收到 CONFIG_UPDATE 消息后，检查指定的通道还不存在，则开始新建过程（参考 `orderer/configupdate/configupdate.go` 文件），构造该应用通道的初始区块。
    - Orderer 首先检查通道应用（Application）配置中的组织都在创建的联盟（Consortium）配置组织中。
    - 之后从系统通道中获取 Orderer 相关的配置，并创建应用通道配置，对应 mod_policy 为系统通道配置中的联盟指定信息。
    - 接下来根据 CONFIG_UPDATE 消息的内容更新获取到的配置信息。所有配置发生变更后版本号都要更新。
    - 最后，创建签名 Proposal 消息（头部类型为 ORDERER_TRANSACTION），发送到系统通道中，完成应用通道的创建过程。
- 客户端利用 gRPC 通道从 Orderer 服务获取到该应用通道的初始区块（具体过程类似 fetch 命令）。
- 客户端将收到的区块写入到本地的 chainID + ".block" 文件。这个文件后续会被需要加入到通道的节点使用。

# 通道加入

主要步骤包括：

- 客户端首先创建一个 ChaincodeSpec 结构，其 input 中的 Args 第一个参数是 CSCC.JoinChain（指定调用配置链码的操作），第二个参数为所加入通道的初始区块。
- 利用 ChaincodeSpec 构造一个 ChaincodeInvocationSpec 结构。
- 利用 ChaincodeInvocationSpec，创建 Proposal 结构并进行签名，channel 头部类型为 CONFIG。
- 客户端通过 gRPC 将 Proposal 签名后发给 Endorser（所操作的 Peer），调用 `ProcessProposal(ctx context.Context, in *SignedProposal, opts ...grpc.CallOption) (*ProposalResponse, error)` 方法进行处理，主要通过配置系统链码进行本地链的初始化工作。
- 初始化完成后，即可收到来自通道内的 Gossip 消息等。

其中，比较重要的数据结构包括 ChaincodeSpec、ChaincodeInvocationSpec、Proposal 等。

# 链码安装过程

链码安装主要包括两个部分：

- 客户端封装安装消息；
- Peer 节点处理请求。

## 客户端封装安装消息

客户端将链码的源码和环境等内容封装为一个链码安装打包文件（Chaincode Install Package，CIP），并传输到指定的 Peer 节点。此过程只需要跟 Peer 节点打交道。

主要步骤包括：

- 首先是构造带签名的提案结构（SignedProposal）。
    - 调用 `InitCmdFactory(isEndorserRequired, isOrdererRequired bool) (*ChaincodeCmdFactory, error)` 方法，初始化 EndoserClient（跟 Peer 通信）、BroadcastClient（跟 Orderer 通信）、Signer（签名操作）等辅助结构体。所有链码子命令都会执行该过程，会根据需求具体初始化不同的结构。
    - 然后根据命令行参数进行解析，判断是根据传入的打包文件来直接读取 ChaincodeDeploymentSpec（CDS）结构，还是根据传入参数从本地链码源代码文件来构造生成。
    - 以本地重新构造情况为例，首先根据命令行中传入的路径、名称等信息，构造生成 ChaincodeSpec（CS）结构。
    - 利用 ChaincodeSpec 结构，结合链码包数据生成一个 ChaincodeDeploymentSpec 结构（chainID 为空），调用本地的 `install(msg proto.Message, cf *ChaincodeCmdFactory) error` 方法。
    - install 方法基于传入的 ChaincodeDeploymentSpec 结构，构造一个对生命周期管理系统链码（LSCC）调用的 ChaincodeSpec 结构，其中，Type 为 ChaincodeSpec_GOLANG，ChaincodeId.Name 为“lscc”，Input 为“install”+ChaincodeDeploymentSpec。进一步地，构造了一个 LSCC 的 ChaincodeInvocationSpec（CIS）结构，对 ChaincodeSpec 结构进行封装。
    - 基于 LSCC 的 ChaincodeInvocationSpec 结构，添加头部结构，生成一个提案（Proposal）结构。其中，通道头部中类型为 ENDORSER_TRANSACTION，TxID 为对随机数+签名实体，进行 Hash。
    - 对 Proposal 进行签名，转化为一个签名后的提案消息结构 SignedProposal。
- 将带签名的提案结构通过 EndorserClient 经由 gRPC 通道发送给 Peer 的 `ProcessProposal(ctx context.Context, in *SignedProposal, opts ...grpc.CallOption) (*ProposalResponse, error)` 接口。
- Peer 模拟运行生命周期链码的调用交易进行处理，检查格式、签名和权限等，通过则保

存到本地文件系统。

下图给出了链码安装过程中最为重要的 SignedProposal 数据结构，该结构对于大部分链码操作命令都是类似的，其中最重要的是 ChannelHeader 结构和 ChaincodeSpec 结构中参数的差异。

链码安装过程中所涉及的数据结构

图 *1.5.13.1 -* 链码安装过程中所涉及的数据结构

# 链码实例化

主要步骤包括：

- 首先，类似链码安装命令，需要创建一个 SignedProposal 消息。注意 instantiate 和 upgrade 支持 policy、escc、vscc 等参数。LSCC 的 ChaincodeSpec 结构中，Input 中包括类型（“deploy”）、通道 ID、ChaincodeDeploymentSpec 结构、背书策略、escc 和 vscc 等。
- 调用 EndorserClient，发送 gRPC 消息，将签名后的 Proposal 发给指定的 Peer 节点（Endorser），调用 `ProcessProposal(ctx context.Context, in *SignedProposal, opts ...grpc.CallOption) (*ProposalResponse, error)` 方法，进行背书处理。节点会模拟运行 LSCC 的调用交易，启动链码容器。实例化成功后会返回 ProposalResponse 消息（其中包括背书签名）。
- 根据 Peer 返回的 ProposalResponse 消息，创建一个 SignedTX（Envelop 结构的交易，带有签名）。
- 使用 BroadcastClient 将交易消息通过 gRPC 通道发给 Orderer，Orderer 会进行全网排序，并广播给 Peer 进行确认提交。

# 链码调用

实现上，基本过程如下：

- 首先，也是要创建一个 SignedProposal 消息。根据传入的各种参数，生成 ChaincodeSpec 结构（其中，Input 为传入的调用参数）。然后，根据 ChaincodeSpec、chainID、签名实体等，生成 ChaincodeInvocationSpec 结构。进而封装生成 Proposal 结构（通道头部中类型为 ENDORSER_TRANSACTION），并进行签名。
- 调用 EndorserClient，发送 gRPC 消息，将签名后的 Proposal 发给指定的 Peer 节点（Endorser），调用 `ProcessProposal(ctx context.Context, in *SignedProposal, opts ...grpc.CallOption) (*ProposalResponse, error)` 方法，进行背书处理。节点会模拟运行链码调用交易，成功后会返回 ProposalResponse 消息（带有背书签名）。
- 根据 Peer 返回的 ProposalResponse 消息，创建一个 SignedTX（Envelop 结构的交易，带有签名）。
- 使用 BroadcastClient 将交易消息通过 gRPC 通道发给 Orderer 进行全网排序并广播给 Peer 进行确认提交。

注意 invoke 是异步操作，invoke 成功只能保证交易已经进入 Orderer 进行排序，但无法保证最终写到账本中（例如交易未通过 Committer 验证而被拒绝）。需要通过 eventHub 或查询方式来进行确认交易是否最终写入到账本上。

# 链码查询

主要过程如下。

- 根据传入的各种参数，最终构造签名提案，通过 endorserClient 发送给指定的 Peer。
- 成功的话，获取到 ProposalResponse，打印出 proposalResp.Response.Payload 内容。

需要注意 invoke 和 query 的区别，query 不需要创建 SignedTx 发送到 Orderer，而且会返回查询的结果。

# bccsp

区块链加密服务提供者（Blockchain Crypto Service Provider），提供一些密码学相关操作的实现，包括 Hash、签名、校验、加解密等。

实现了软件机制（sw）和硬件机制（pkcs11）。

bccsp 主要支持 MSP 的相关调用。

# factory

提供工厂模式支持，包括若干类型的 BCCSP 工厂实现。

通用工厂接口为：

```
type BCCSPFactory interface {

    // Name returns the name of this factory
    Name() string

    // Get returns an instance of BCCSP using opts.
    Get(opts *FactoryOpts) (bccsp.BCCSP, error)
}
```

实现了两个工厂结构：

- SWFactory：软件 bccsp 的工厂；
- PKCS11Factory：基于高安全模块的实现。

# factory.go

# nopkcs11.go

## opts.go

# pkcs11.go

# pkcs11factory.go

# pluginfactory.go

# swfactory.go

# mocks

# mocks.go

# pkcs11

# conf.go

# ecdsa.go

# ecdsakey.go

# impl.go

# pkcs11.go

# signer

# signer.go

# sw

# mocks

## mocks.go

# aes.go

# aeskey.go

# conf.go

# dummyks.go

# ecdsa.go

# ecdsakey.go

# fileks.go

# hash.go

# impl.go

# internals.go

# keyderiv.go

# keygen.go

# keyimport.go

# rsa.go

# rsakey.go

# utils

# errs.go

# io.go

## keys.go

# slice.go

# x509.go

# aesopts.go

AES 算法相关选项结构。

# bccsp.go

- BCCSP 接口：定义密码学相关操作，包括加解密、签名和验证、签名、Hash、Key 的生命周期管理等方法。

```
type BCCSP interface {

    // KeyGen generates a key using opts.
    KeyGen(opts KeyGenOpts) (k Key, err error)

    // KeyDeriv derives a key from k using opts.
    // The opts argument should be appropriate for the primitive used.
    KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)

    // KeyImport imports a key from its raw representation using opts.
    // The opts argument should be appropriate for the primitive used.
    KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)

    // GetKey returns the key this CSP associates to
    // the Subject Key Identifier ski.
    GetKey(ski []byte) (k Key, err error)

    // Hash hashes messages msg using options opts.
    // If opts is nil, the default hash function will be used.
    Hash(msg []byte, opts HashOpts) (hash []byte, err error)

    // GetHash returns and instance of hash.Hash using options opts.
    // If opts is nil, the default hash function will be returned.
    GetHash(opts HashOpts) (h hash.Hash, err error)

    // Sign signs digest using key k.
    // The opts argument should be appropriate for the algorithm used.
    //
    // Note that when a signature of a hash of a larger message is needed,
    // the caller is responsible for hashing the larger message and passing
    // the hash (as digest).
    Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)

    // Verify verifies signature against key k and digest
    // The opts argument should be appropriate for the algorithm used.
    Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)

    // Encrypt encrypts plaintext using key k.
    // The opts argument should be appropriate for the algorithm used.
    Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error
)

    // Decrypt decrypts ciphertext using key k.
    // The opts argument should be appropriate for the algorithm used.
    Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error
)
}
```

# ecdsaopts.go

ECDSA 算法相关选项结构。

# hashopts.go

Hash 算法（SHA）相关选项结构。

# keystore.go

定义 KeyStore 接口，存储秘钥。

```
type KeyStore interface {

    // ReadOnly returns true if this KeyStore is read only, false otherwise.
    // If ReadOnly is true then StoreKey will fail.
    ReadOnly() bool

    // GetKey returns a key object whose SKI is the one passed.
    GetKey(ski []byte) (k Key, err error)

    // StoreKey stores the key k in this KeyStore.
    // If this KeyStore is read only then the method will fail.
    StoreKey(k Key) (err error)
}
```

# opts.go

提供一些基础的密码学选项，包括常见算法名称、秘钥生成参数等。

# rsaopts.go

RSA 算法相关选项结构，包括RSA2048、RSA3072、RSA4096 算法秘钥生成选项等。

# bddtests

行为驱动测试（Behaviour Driven Development）相关代码。

# common

# common_pb2.py

# common_pb2_grpc.py

# configtx_pb2.py

# configtx_pb2_grpc.py

# configuration_pb2.py

# configuration_pb2_grpc.py

# ledger_pb2.py

# ledger_pb2_grpc.py

# policies_pb2.py

# policies_pb2_grpc.py

# features

对 endorser 和 orderer 服务的功能进行简单测试。

包括 BDD 测试的脚本。

# bootstrap.feature

# endorser.feature

# orderer.feature

# msp

# identities_pb2.py

# identities_pb2_grpc.py

# msp_config_pb2.py

# msp_config_pb2_grpc.py

# msp_principal_pb2.py

# msp_principal_pb2_grpc.py

# orderer

# ab_pb2.py

# ab_pb2_grpc.py

# configuration_pb2.py

# configuration_pb2_grpc.py

# kafka_pb2.py

# kafka_pb2_grpc.py

# peer

# admin_pb2.py

# admin_pb2_grpc.py

# chaincode_event_pb2.py

# chaincode_event_pb2_grpc.py

# chaincode_pb2.py

# chaincode_pb2_grpc.py

# chaincode_shim_pb2.py

# chaincode_shim_pb2_grpc.py

# configuration_pb2.py

# configuration_pb2_grpc.py

# events_pb2.py

# events_pb2_grpc.py

# peer_pb2.py

# peer_pb2_grpc.py

# proposal_pb2.py

# proposal_pb2_grpc.py

# proposal_response_pb2.py

# proposal_response_pb2_grpc.py

# query_pb2.py

# query_pb2_grpc.py

# transaction_pb2.py

# transaction_pb2_grpc.py

# scripts

启动 peer 节点的脚本。

# wait-for-it.sh

等待某给定服务的端口可用（服务启动起来），否则不断探测。

# steps

# bdd_grpc_util.py

# bdd_test_util.py

# bootstrap_impl.py

# bootstrap_util.py

# compose.py

# contexthelper.py

# coverage.py

# docgen.py

# endorser_impl.py

# endorser_util.py

# orderer_impl.py

# orderer_util.py

# templates

# html

## appendix-py.html

# cli.html

## composition-py.html

## directory-py.html

## directory.html

## error.html

## graph.html

## header.html

## main.html

## org-py.html

# org.html

## protobuf-py.html

# protobuf.html

## report.css

## scenario.html

## step.html

## tag.html

## user.html

# chaincode.go

# compose.go

# conn.go

# context.go

# context_bootstrap.go

# context_endorser.go

# dc-base.yml

# dc-orderer-base.yml

# dc-orderer-kafka-base.yml

# dc-orderer-kafka.yml

# dc-peer-base.yml

# dc-peer-couchdb.yml

# docker.go

# environment.py

# tlsca.cert

# tlsca.priv

# users.go

# util.go

# common

一些通用的功能模块。

定义相应的接口和结构。

# attrmgr 包

# attrmgr.go

# capabilities 包

# application.go

# capabilities.go

# channel.go

## orderer.go

# cauthdsl

实现 Policy 相关的数据结构和方法。

fabric 中，策略主要分为签名策略（SignaturePolicy）和隐式策略（ImplicitMetaPolicy）两种。

最常见的签名策略采用 SignaturePolicyEnvelope 结构来表达。定义在 protos/common/policies.go中。

```
type SignaturePolicyEnvelope struct {
    Version    int32                   `protobuf:"varint,1,opt,name=version" json:"version,omitempty"`
    Rule       *SignaturePolicy        `protobuf:"bytes,2,opt,name=rule" json:"rule,omitempty"`
    Identities []*common1.MSPPrincipal `protobuf:"bytes,3,rep,name=identities" json:"identities,omitempty"`
}
```

签名策略的规则为检查签名是否满足指定的 principal（特定个体、身份等）。

更通用的 Implicit 策略是一个递归结构，可以指定依赖其它策略，最终底层为签名策略。

# cauthdsl.go

主要实现了 `func compile(policy *cb.SignaturePolicy, identities []*mb.MSPPrincipal, deserializer msp.IdentityDeserializer) (func([]*cb.SignedData, []bool) bool, error)` 方法。

该方法根据传入的策略结构，返回一个函数 `func([]*cb.SignedData, []bool) bool, error`，作为策略 evaluate 方法。

# cauthdsl_builder.go

定义一些常见的策略。

包括：

- AcceptAllPolicy
- RejectAllPolicy

另外，提供了一些常用的构造方法，包括

- Envelope(policy *cb.SignaturePolicy, identities [][]byte)* cb.SignaturePolicyEnvelope：快速构造一个策略信封结构。

一些策略构造方法：

- SignedBy(index int32) *cb.SignaturePolicy：构造一条指定签名者的签名策略。
- SignedByMspMember(mspId string) *cb.SignaturePolicyEnvelope：指定一条指定签名身份为某 MSP 成员（至少一个）的签名策略。
- SignedByMspAdmin(mspId string) *cb.SignaturePolicyEnvelope：指定一条指定签名身份为某 MSP 管理员（至少一个）的签名策略。
- SignedByAnyMember(ids []string) *cb.SignaturePolicyEnvelope：被给定组织列表中任意成员签名。
- SignedByAnyAdmin(ids []string) *cb.SignaturePolicyEnvelope：被给定组织列表中任意管理员签名。

一些策略的组合逻辑（与、或、集合中的若干个）方法：

- And(lhs, rhs *cb.SignaturePolicy)* cb.SignaturePolicy：两个策略的与组合。
- Or(lhs, rhs *cb.SignaturePolicy)* cb.SignaturePolicy：两个策略的或组合。
- NOutOf(n int32, policies []*cb.SignaturePolicy)* cb.SignaturePolicy：集合中的至少若干个。

# policy.go

实现上，一条 policy 结构实现一个 evalutor 方法。

```
type policy struct {
    evaluator func([]*cb.SignedData, []bool) bool
}
```

对外提供的方法是 Evaluate() 方法，调用了 evaluator() 私有方法。

```
func (p *policy) Evaluate(signatureSet []*cb.SignedData) error {
    if p == nil {
        return fmt.Errorf("No such policy")
    }

    ok := p.evaluator(signatureSet, make([]bool, len(signatureSet)))
    if !ok {
        return errors.New("Failed to authenticate policy")
    }
    return nil
}
```

同时，提供了 NewPolicyProvider(deserializer msp.IdentityDeserializer) policies.Provider 方法，作为工厂方法。

# policy_util.go

# policyparser.go

主要提供了 `FromString(policy string) (*common.SignaturePolicyEnvelope, error)` 方法，将字符串指定的策略生成 Policy 信封结构。

# channelconfig 包

## api.go

## application.go

# applicationorg.go

# bundle.go

# bundlesource.go

# channel.go

# consortium.go

## consortiums.go

# logsanitychecks.go

# msp.go

## orderer.go

# organization.go

# standardvalues.go

## util.go

# configtx

负责 config transaction（新建或更新）相关的数据结构和处理。

## test

## helper.go

# compare.go

# configmap.go

# configtx.go

# update.go

# util.go

# validator.go

# crypto

主要定义了 LocalSigner 接口。

# random.go

## signer.go

# errors

错误相关代码。

# codes.go

# errors.go

# flogging

# grpclogger.go

# logging.go

# genesis

提供初始区块的工厂接口和实现，提供生成的方法。

# genesis.go

# ledger

账本结构的接口和实现。

# blkstorage 包

链结构存放在本地文件系统上。由 blkstorage 包负责提供底层的支持。

# fsblkstorage

## blockstorage.go

# testutil

## test_helper.go

## test_util.go

# util

## leveldbhelper

## ioutil.go

## protobuf_util.go

## util.go

# ledger_interface.go

# localmsp

## signer.go

# metadata

# metadata.go

# metrics 包

## server.go

# tally_provider.go

# types.go

# mocks

对主要模块们提供 mock 包。

# config

## application.go

## channel.go

## orderer.go

## resources.go

# configtx

# configtx.go

# crypto

## localsigner.go

# ledger

## queryexecutor.go

# msp

## noopmsp.go

# peer

## mockccstream.go

## mockpeerccsupport.go

# policies

## policies.go

# scc

## sccprovider.go

# policies

策略的处理。

策略目前包括 ImplicitMetaPolicy 和 SignaturePolicy 两种。

- ImplicitMetaPolicy：基于其它策略（往往是子空间中策略）的结果来判断。
- SignaturePolicy：基于签名的策略，签名集合中某种特定组合符合特定条件。如至少 3 个签名，其中 1 个管理员，2 个成员。

```
type implicitMetaPolicy struct {
    conf        *cb.ImplicitMetaPolicy
    threshold   int
    subPolicies []Policy
}
```

# implicitmeta.go

# implicitmeta_util.go

# policy.go

主要定义了三个策略相关接口和实现了这三个接口的结构。

- ChannelPolicyManagerGetter：获取指定通道的策略管理器。
- Manager：一个通道相关策略的管理器。
- Proposer：负责处理策略更新的相关操作。

ManagerImpl 结构实现了这三个接口，成为策略相关操作的主要入口结构。

```
type ManagerImpl struct {
    parent        *ManagerImpl
    basePath      string
    fqPrefix      string
    providers     map[int32]Provider
    config        *policyConfig
    pendingConfig map[interface{}]*policyConfig
    pendingLock   sync.RWMutex

    // SuppressSanityLogMessages when set to true will prevent the sanity checking log
    // messages.  Useful for novel cases like channel templates
    SuppressSanityLogMessages bool
}
```

## util.go

# resourcesconfig 包

## apis.go

# bundle.go

# bundlesource.go

# chaincode.go

## chaincodes.go

# peerpolicies.go

# policyrouter.go

# resources.go

# resourcesconfig.go

# tools

几个有用的工具，包括 configtxlator 和 cryptogen 等。

# configtxgen

configtxgen 工具提供了跟配置交易相关的功能，包括生成创世区块、生成创建通道的交易、生成更新锚点配置的交易等。

## encoder 包

## localconfig

## metadata 包

# configtxgen

configtxgen 工具是一个很重要的离线辅助工具，它的主要功能有如下几个：

* 生成启动 orderer 需要的初始区块，并检查区块内容；
* 生成创建 channel 需要的配置交易，并检查交易内容；
* 生成锚点 Peer 的更新配置信息。

默认情况下，configtxgen 工具会依次尝试从 `$FABRIC_CFG_PATH` 环境变量指定的路径，当前路径和 `/etc/hyperledger/fabric` 路径下查找 configtx.yaml 配置文件并读入。

main.go 文件为入口。

核心代码十分简单：

* 首先通过 `factory.InitFactories(nil)` 读入 BCCSP 配置；
* 通过 `config := genesisconfig.Load(profile)` 读入 configtx.yaml 配置文件；
* 然后根据指定的参数分别进行相应的操作。

```
func main() {
    var outputBlock, outputChannelCreateTx, profile, channelID, inspectBlock, inspectChannelCreateTx, outputAnchorPeersUpdate, asOrg string

    flag.StringVar(&outputBlock, "outputBlock", "", "The path to write the genesis block to (if set)")
    flag.StringVar(&channelID, "channelID", provisional.TestChainID, "The channel ID to use in the configtx")
    flag.StringVar(&outputChannelCreateTx, "outputCreateChannelTx", "", "The path to write a channel creation configtx to (if set)")
    flag.StringVar(&profile, "profile", genesisconfig.SampleInsecureProfile, "The profile from configtx.yaml to use for generation.")
    flag.StringVar(&inspectBlock, "inspectBlock", "", "Prints the configuration contained in the block at the specified path")
    flag.StringVar(&inspectChannelCreateTx, "inspectChannelCreateTx", "", "Prints the configuration contained in the transaction at the specified path")
    flag.StringVar(&outputAnchorPeersUpdate, "outputAnchorPeersUpdate", "", "Creates an config update to update an anchor peer (works only with the default channel creation, and only for the first update)")
    flag.StringVar(&asOrg, "asOrg", "", "Performs the config generation as a particular organization, only including values in the write set that org (likely) has privilege to set")

    flag.Parse()

    logging.SetLevel(logging.INFO, "")

    logger.Info("Loading configuration")
    factory.InitFactories(nil)
    config := genesisconfig.Load(profile)
```

```
    if outputBlock != "" {
        if err := doOutputBlock(config, channelID, outputBlock); err != nil {
            logger.Fatalf("Error on outputBlock: %s", err)
        }
    }

    if outputChannelCreateTx != "" {
        if err := doOutputChannelCreateTx(config, channelID, outputChannelCreateTx); e
rr != nil {
            logger.Fatalf("Error on outputChannelCreateTx: %s", err)
        }
    }

    if inspectBlock != "" {
        if err := doInspectBlock(inspectBlock); err != nil {
            logger.Fatalf("Error on inspectBlock: %s", err)
        }
    }

    if inspectChannelCreateTx != "" {
        if err := doInspectChannelCreateTx(inspectChannelCreateTx); err != nil {
            logger.Fatalf("Error on inspectChannelCreateTx: %s", err)
        }
    }

    if outputAnchorPeersUpdate != "" {
        if err := doOutputAnchorPeersUpdate(config, channelID, outputAnchorPeersUpdate
, asOrg); err != nil {
            logger.Fatalf("Error on inspectChannelCreateTx: %s", err)
        }
    }
}
```

### doOutputBlock

生成初始区块，存放到指定路径。

- 首先调用 provisional.New(config) 根据读入的配置生成启动参数结构。
- 调用 genesisBlock := pgen.GenesisBlockForChannel(channelID) 生成指定通道的区块，核心数据是一个 ConfigEnvelope 结构。
- 将区块文件写到本地。

### doOutputChannelCreateTx

生成创建新通道的交易，存放到指定路径。

- 获取签名实体信息。
- 生成创建新通道的交易结构。
- 写入到本地文件。

#### doInspectBlock

读入区块内容，并解析为 ConfigEnvelope 结构，打印出来。

#### doInspectChannelCreateTx

读入交易文件内容，并解析为 ConfigUpdateEnvelope 结构，打印出来。

#### doOutputAnchorPeersUpdate

生成一个锚点 Peer 更新的 ConfigUpdateEnvelope，并存储到指定路径。

# configtxlator

## metadata

## rest

## sanitycheck

## update

## main.go

# cryptogen

在 Fabric 中，通过证书和密钥来管理身份，经常需要进行证书生成和配置操作。

使用 cryptogen 工具，可以快速根据配置自动生成所需要的密钥和证书文件，或者查看配置模板信息。

入口代码在 main.go。

## ca

# csp

## metadata

## msp

## main.go

核心代码十分简单。

```
//command line flags
var (
    app = kingpin.New("cryptogen", "Utility for generating Hyperledger Fabric key mate
rial")

    gen        = app.Command("generate", "Generate key material")
    outputDir  = gen.Flag("output", "The output directory in which to place artifacts"
).Default("crypto-config").String()
    configFile = gen.Flag("config", "The configuration template to use").File()

    showtemplate = app.Command("showtemplate", "Show the default configuration templat
e")
)

func main() {
    kingpin.Version("0.0.1")
    switch kingpin.MustParse(app.Parse(os.Args[1:])) {

    // "generate" command
    case gen.FullCommand():
        generate()

    // "showtemplate" command
    case showtemplate.FullCommand():
        fmt.Print(defaultConfig)
        os.Exit(0)
    }

}
```

### generate

主要调用 generate() 方法。

- 首先调用 getConfig()读取本地配置或使用默认配置。
- 使用 renderOrgSpec() 和 generatePeerOrg()方法依次生成每个 peer 类型的组织。
- 使用 renderOrgSpec() 和 generateOrdererOrg()方法依次生成每个 orderer 类型的组织。

renderOrgSpec()负责将模板中具体组织数据解析出来。

#### peer 组织

generatePeerOrg（）会具体生成每个 peer 类型的组织，结构如下。

peerOrganizations：

- org1：存放第一个组织的相关材料，每个组织会生成单独的根证书。
    - ca：存放组织的根证书和对应的私钥文件，默认采用 EC 算法，证书为自签名。组织内的实体将基于该根证书作为相同的证书根。
    - msp：存放代表该组织的身份信息
        - admincerts：组织管理员的身份验证证书。
        - cacerts：组织的根证书。
        - keystore：组织的私钥文件，用来签名。
        - signcerts：组织的签名验证证书。
    - peers：存放该组织下的所有 peer 节点
        - peer1：第一个 peer 的信息，包括 msp 证书和 tls 证书两类。
            - msp：
                - admincerts：组织管理员的身份验证证书。
                - cacerts：存放组织的根证书。
                - keystore：本节点的身份私钥，用来签名。
                - signcerts：验证本节点签名的证书，被组织根证书签名。
            - tls：存放 tls 相关的证书和私钥
                - ca.crt：组织的根证书。
                - server.crt：验证本节点签名的证书，被组织根证书签名。
                - server.key：本节点的身份私钥，用来签名。
    - users：存放属于该组织的用户的实体。
        - Admin：管理员用户的信息，包括 msp 证书和 tls 证书两类。
            - msp：
                - admincerts：组织根证书作为管理者身份验证证书。
                - cacerts：存放组织的根证书。
                - keystore：本用户的身份私钥，用来签名。
                - signcerts：管理员用户的身份验证证书，被组织根证书签名。
            - tls：存放 tls 相关的证书和私钥
                - ca.crt：组织的根证书。
                - server.crt：管理员的用户身份验证证书，被组织根证书签名。
                - server.key：管理员用户的身份私钥，用来签名。
        - User1：第一个用户的信息，包括 msp 证书和 tls 证书两类。
            - msp：
                - admincerts：组织根证书作为管理者身份验证证书。
                - cacerts：存放组织的根证书。
                - keystore：本用户的身份私钥，用来签名。
                - signcerts：验证本用户签名的身份证书，被组织根证书签名。
            - tls：存放 tls 相关的证书和私钥
                - ca.crt：组织的根证书。
                - server.crt：验证用户签名的身份证书，被组织根证书签名。

        - server.key：用户的身份私钥，用来签名。
- org2
    - ...

注意每个实体（组织、节点、用户）最终都会拥有 MSP 来代表身份信息，其中包括三种证书：管理员身份的验证证书、实体信任的 CA 的根证书、自身身份验证（检查签名）的证书。此外，还包括对应身份验证的签名用的私钥。

GenerateVerifyingMSP()会生成组织的 MSP 信息。

generateNodes 会针对每个实体生成所需要的 msp 和 tls 信息，证书都会被组织根证书签名，信任的根都是组织 CA 证书。

#### orderer 组织

跟 peer 组织类似过程。

## idemixgen 包

# idemixca 包

# idemixgen.go

# protolator

# testprotos

## api.go

## dynamic.go

# json.go

## nested.go

## statically_opaque.go

## variably_opaque.go

# util 包

一些辅助方法，包括计算各种 Hash，获取默认链和组织 ID 等。

# utils.go

# viperutil

# config_util.go

# core

大部分核心实现代码都在本包下。其它包的代码封装上层接口，最终调用本包内代码。

# aclmgmt 包

## mocks 包

## mocks.go

# aclmgmt.go

# aclmgmtimpl.go

# defaultaclprovider.go

# chaincode

chaincode 相关，包括生成 chaincode 镜像，支持对 chaincode 的调用、查询等。

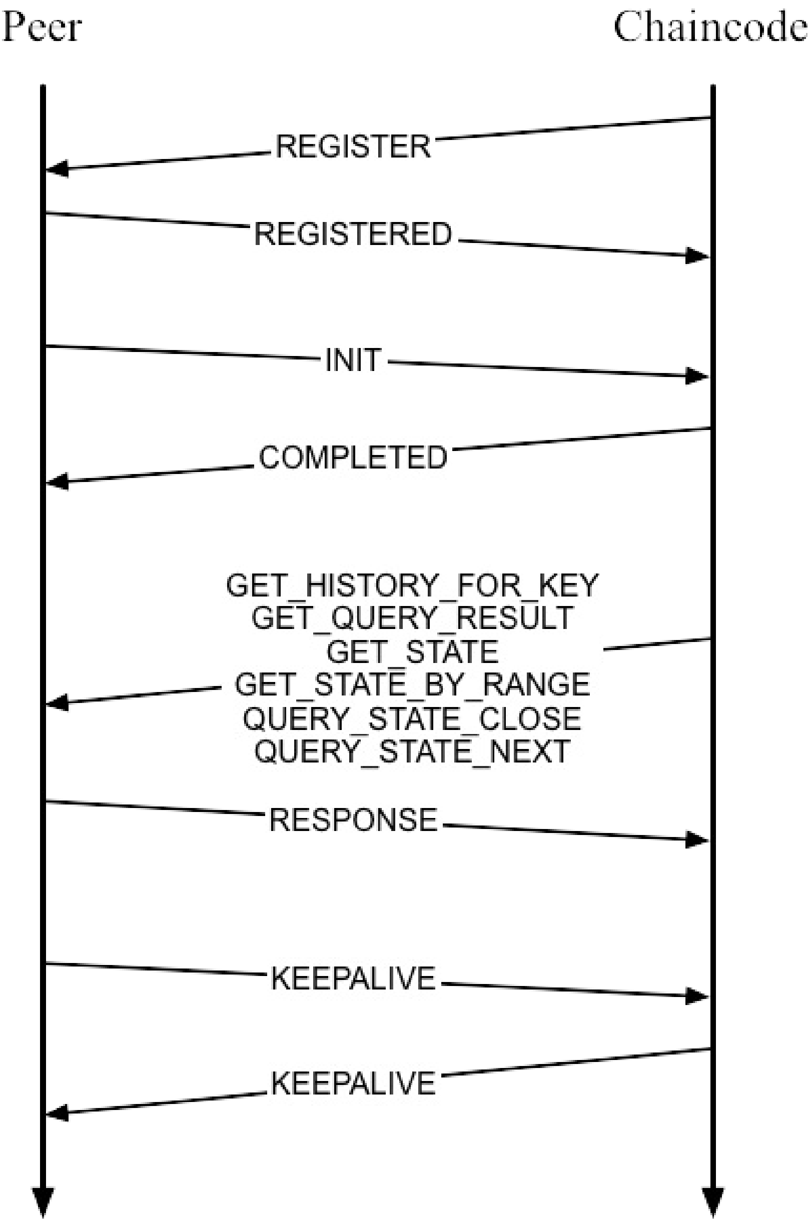
包内代码分两部分，根目录下是 Peer 侧代码，比较核心的结构包括：

- ChaincodeSupport：通过调用 vmc 驱动来支持对 chaincode 容器的管理，包括启动容器、注册、执行合约等；
- Handler：cc 容器启动后，会发送注册消息让 peer 创建一个 handler 结构，并进入主循环响应消息。peer 侧通过一个状态机来维护对于 chaincode 各种消息的响应，利用 before、after 等触发条件。

下面 shim 包则提供 Chaincode 跟账本结构打交道的中间层。

- ChaincodeStub：chaincode 中代码通过该结构提供的方法来修改账本状态；
- Handler：chaincode 一侧用状态机来跟踪 shim 相关事件。

platforms 包提供具体的对 Chaincode 运行类型的支持，包括 golang、java、car 等。例如在部署的时候完成打包任务。


Peer
Chaincode
REGISTER
REGISTERED
INIT
COMPLETED
GET_HISTORY_FOR_KEY
GET_QUERY_RESULT
GET_STATE
GET_STATE_BY_RANGE
QUERY_STATE_CLOSE
QUERY_STATE_NEXT
RESPONSE
KEEPALIVE
KEEPALIVE

# accesscontrol 包

## access.go

## ca.go

## interceptor.go

## key.go

## mapper.go

# lib 包

## cid 包

# platforms

这里面主要是生成 chaincode 容器镜像时候，进行打包的一些方法类。

Fabric v1.1支持以下几种类型：car、golang、node、java。

定义了接口 Platform。

```
type Platform interface {
    ValidateSpec(spec *pb.ChaincodeSpec) error
    ValidateDeploymentSpec(spec *pb.ChaincodeDeploymentSpec) error
    GetDeploymentPayload(spec *pb.ChaincodeSpec) ([]byte, error)
    GenerateDockerfile(spec *pb.ChaincodeDeploymentSpec) (string, error)
    GenerateDockerBuild(spec *pb.ChaincodeDeploymentSpec, tw *tar.Writer) error
}
```

ValidateSpec 用来对一个 chaincodeSpec 进行检查； ValidateDeploymentSpec 用来检查部署时(对应sdk的Client.install，lscc的install Transaction)的deploymentSpec, 这个deploymentSpec是由SDK构造并传入的 GetDeploymentPayload 按照部署时的chaincodePath，将chaincodePath目录下的所有文件打包到一个tar包里 GenerateDockerfile 根据chaincode的deploymentSpec，确定chaincode使用的runtime（go和node使用hyperledger/fabric-ccenv, java使用hyperledger/fabric-javaenv）,构造build dockerimage的Dockerfile GenerateDockerBuild 编译chaincode，（java 对应的是gradle build， node对应的是npm install）

所有的类型都实现了Platform接口

## car

car 是打包好的 chaincode 的一种格式，可以参考 chaincodetool。

# golang

golang 格式的 chaincode。

## java

java 格式的 chaincode。

# Node Chaincode Platform

### ValidateDeploymentSpec

对于node.js的cc来说，必须要有一个 `package.json` 文件在chaincode的根目录下面，在启动这个chaincode的时候， 会使用 `npm install` 和 `npm start` 命令，所以在使用sdk部署chaincode的时候，会对chaincode进行验证，确认是 否有 `package.json` 文件在根目录下面。

### GetDeploymentPayload

将chaincodePath目录下除了 `node_modules` 之外的所有文件打进一个tar包

### GenerateDockerfile

使用hyperledger/fabric-ccenv构造chaincode的运行image，可以看到在这一步里面是将chaincode复制到 `/usr/local/src` 目录下了, 在chaincode部署(instantiate)成功时，可以在 `/usr/local/src` 看到部署的cc

### GenerateDockerBuild

build `binpackage.tar` , `binpackage.tar` 包含了所有的chaincode源码以及依赖库(node_modules) `cp -R /chaincode/input/src/* /chaincode/output && cd /chaincode/output && npm install -production`

## util

# platforms.go

定义抽象的 platform 接口。

# shim

shim 包可以让 chaincode 代码跟 ledger 进行交互。

比较重要的 ChaincodeStub 结构，提供了用户 chaincode 代码跟 ledger 进行交互的一系列方法，例如 PutState 与 GetState 来写入和查询链上键值对的状态。

用户链码 --> ChaincodeStub --> Handler -----ChaincodeMessage -----> Peer

## ext 包

## java

## chaincode.go

提供 ChaincodeStub 结构，支持一系列对账本进行操作的方法（如 GetState、PutState、DelState 等），这些方法用户可以直接在链码中进行调用。

```
type ChaincodeStub struct {
    TxID           string
    chaincodeEvent *pb.ChaincodeEvent
    args           [][]byte
    handler        *Handler
    signedProposal *pb.SignedProposal
    proposal       *pb.Proposal

    // Additional fields extracted from the signedProposal
    creator   []byte
    transient map[string][]byte
    binding   []byte
}
```

stub 会进一步调用本地的 Handler 结构提供的方法，主要过程都是封装为 ChaincodeMessage，发给 peer 节点进行指定的操作。

## chaincode_experimental.go

## handler.go

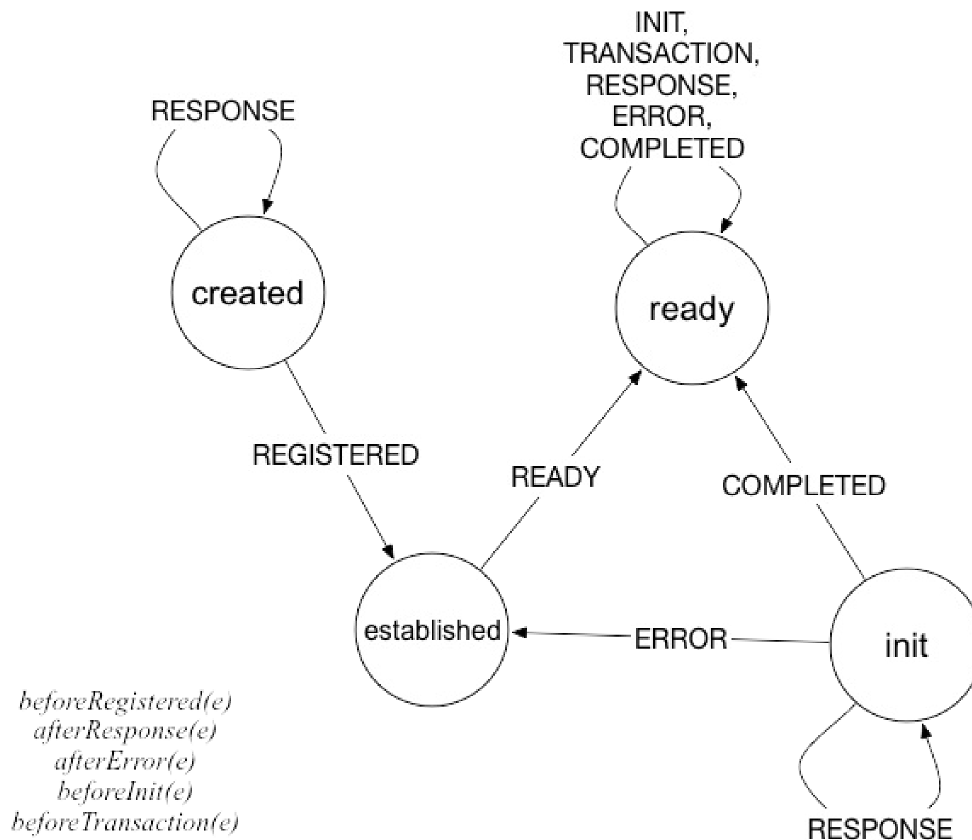
最主要的是实现了 Handler 结构体，通过各种 handleXXX 方法具体实现来自 Chaincode 接口中定义的的各种对账本的操作。

```
type Handler struct {
    sync.RWMutex
    //shim to peer grpc serializer. User only in serialSend
    serialLock sync.Mutex
    To         string
    ChatStream PeerChaincodeStream
    FSM        *fsm.FSM
    cc         Chaincode
    // Multiple queries (and one transaction) with different txids can be executing in
 parallel for this chaincode
    // responseChannel is the channel on which responses are communicated by the shim
to the chaincodeStub.
    responseChannel map[string]chan pb.ChaincodeMessage
    nextState       chan *nextStateInfo
}
```

成员主要包括：

- ChatStream：跟 Peer 进行通信的 grpc 流。
- FSM：最重要的事件处理状态机，根据收到不同事件调用不同方法。
- cc：所面向的链码。
- responseChannel：本地 chan。字典结构，key 是 TxID，value 里面可以放上一些消息，供调用者后面使用。
- nextState：本地 chan，可以存放下一步要进行的操作和数据。

#### FSM




FSM 的初始化在 newChaincodeHandler() 方法中。

下面是 FSM 可能会触发的方法，每个方法首先都会从 e.Args[0] 尝试解析 ChaincodeMessage 结构，如果失败则退出，成功则继续。

- beforeRegistered(e)：收到了注册到 peer 成功的消息，不进行任何操作。
- afterResponse(e)：将消息放到 responseChannel 中。
- afterError(e)：将消息放到 responseChannel 中。
- beforeInit(e)：收到初始化请求 INIT 消息。解析消息后调用 Handler.handleInit()方法进行处理。该方法从消息 Payload 中解析出 ChaincodeInput 结构，利用这些信息，新建 stub 结构，并调用 stub.init 方法对 stub 进行初始化（配置 TxID、args、handler、signedProposal、creator、transient、binding 等成员）。之后，调用 Handler 结构成员 chaincode 结构的 Init 方法（由用户编写）。将收到的结果构造一个 COMPLETED ChaincodeMessage，放到 nextState 里面待发送。
- beforeTransaction(e)：收到发起交易的请求 TRANSACTION 消息。解析消息后调用 Handler.handleTransaction()方法进行处理。该方法从消息 Payload 中解析出 ChaincodeInput 结构，利用这些信息，新建 stub 结构，并调用 stub.init 方法对 stub 进行初始化（配置 TxID、args、handler、signedProposal、creator、transient、binding 等成员）。之后，调用 Handler 结构成员 chaincode 结构的 Invoke 方法（由用户编

写）。将收到的结果构造一个 COMPLETED ChaincodeMessage，放到 nextState 里面待发送。

## inprocstream.go

## interfaces_experimental.go

## interfaces_stable.go

## mockstub.go

## response.go

# testdata

## server1.key

## server1.pem

# ccproviderimpl.go

ccProviderImpl 结构封装了对链码操作的上层方法。

```
type ccProviderImpl struct {
    txsim ledger.TxSimulator
}
```

# chaincode_support.go

主要实现 ChaincodeSupport 结构，这是 peer 侧对链码支持的主要数据结构。peer 启动后，会初始化一个该结构体。

```
type ChaincodeSupport struct {
    auth              accesscontrol.Authenticator
    runningChaincodes *runningChaincodes
    peerAddress       string
    ccStartupTimeout  time.Duration
    peerNetworkID     string
    peerID            string
    peerTLSCertFile   string
    peerTLSKeyFile    string
    peerTLSSvrHostOrd string
    keepalive         time.Duration
    chaincodeLogLevel string
    shimLogLevel      string
    logFormat         string
    executetimeout    time.Duration
    userRunsCC        bool
    peerTLS           bool
}
```

其中，runningChaincodes 结构十分重要，其中通过 chaincodeMap 字典结构维护所有关联链码容器的处理句柄。每一个启动后的链码容器都会对应到这里面的一个句柄。

```
// runningChaincodes contains maps of chaincodeIDs to their chaincodeRTEs
type runningChaincodes struct {
    sync.RWMutex
    // chaincode environment for each chaincode
    chaincodeMap map[string]*chaincodeRTEnv

    //mark the starting of launch of a chaincode so multiple requests
    //do not attempt to start the chaincode at the same time
    launchStarted map[string]bool
}

type chaincodeRTEnv struct {
    handler *Handler
}
```

方法主要包括：

- `Execute(ctxt context.Context, cccid *ccprovider.CCContext, msg *pb.ChaincodeMessage, timeout time.Duration) (*pb.ChaincodeMessage, error)`：在链码侧执行一个交易。
- `HandleChaincodeStream(ctxt context.Context, stream ccintf.ChaincodeStream) error`：响

应链码容器消息流。

- `Launch(context context.Context, cccid *ccprovider.CCContext, spec interface{}) (*pb.ChaincodeID, *pb.ChaincodeInput, error)` ：启动一个链码容器，等待它完成注册。
- `Register(stream pb.ChaincodeSupport_RegisterServer) error` ：创建并初始化 peer 端的 chaincode处理的 Handler。
- `Stop(context context.Context, cccid *ccprovider.CCContext, cds *pb.ChaincodeDeploymentSpec) error` ：停止一个链码容器。

# chaincodeexec.go

支持从 LSCC 中获取 CDS 和 链码数据。通过调用 ExecuteChaincode() 来实现。

- GetCDSFromLSCC()
- GetChaincodeDataFromLSCC()

# ExecuteChaincode

- 创建简单的 ChaincodeInvocationSpec，只包含链码名字和Args。
- 核心：调用 Execute()方法（见core/chaincode/exectransaction.go），返回响应response和事件event。
- 返回response、event。

# chaincodetest.yaml

# exectransaction.go

提供 `func Execute(ctxt context.Context, cccid *ccprovider.CCContext, spec interface{}) (*pb.Response, *pb.ChaincodeEvent, error)` 方法。

主要过程：

- 提起 ChaincodeSupport，Launch()（检查链码容器是否已经提起来了，如果没有则提起 chaincode，等待 register状态，转换为ready状态）。
- 通过 ChaincodeSupport 来执行，Execute()（检查链码容器是否运行，handler.sendExecuteMessage()，监听并返回消息 response，类型为 ChaincodeMessage）。
- 检查response.Type 是否为 COMPLETED
- 返回解码后的 response.Payload、response.ChaincodeEvent。

# handler.go

主要提供：

- Handler 结构体：对所关联的链码容器进行相应，内部有一个状态机。
- HandleChaincodeStream() 方法：对外提供初始化的 Handler 结构体，并进入循环，不断接收来自链码容器的消息。

## Handler 结构体

Peer 侧会为每一个 chaincode 维护一个 Handler 结构，具体响应所绑定的 chaincode 容器过来的各种消息，通过内部状态机进行处理。

Handler 结构实现了 MessageHandler 接口，主要提供一个 `HandleMessage(msg *pb.ChaincodeMessage) error` 方法，作为处理各个消息的入口方法。

```
type Handler struct {
    sync.RWMutex
    //peer to shim grpc serializer. User only in serialSend
    serialLock  sync.Mutex
    ChatStream  ccintf.ChaincodeStream
    FSM         *fsm.FSM
    ChaincodeID *pb.ChaincodeID
    ccInstance  *sysccprovider.ChaincodeInstance

    chaincodeSupport *ChaincodeSupport
    registered       bool
    readyNotify      chan bool
    // Map of tx txid to either invoke tx. Each tx will be
    // added prior to execute and remove when done execute
    txCtxs map[string]*transactionContext

    txidMap map[string]bool

    // used to do Send after making sure the state transition is complete
    nextState chan *nextStateInfo

    policyChecker policy.PolicyChecker
}
```

chaincode 容器启动后，会调用到服务端的 Register() 方法，该方法进一步调用到 HandleChaincodeStream()，创建 Handler 结构体，进入接收消息循环。

```
// HandleChaincodeStream Main loop for handling the associated Chaincode stream
func HandleChaincodeStream(chaincodeSupport *ChaincodeSupport, ctxt context.Context, s
tream ccintf.ChaincodeStream) error {
    deadline, ok := ctxt.Deadline()
    chaincodeLogger.Debugf("Current context deadline = %s, ok = %v", deadline, ok)
    handler := newChaincodeSupportHandler(chaincodeSupport, stream)
    return handler.processStream()
}
```

newChaincodeSupportHandler 方法中会初始化 FSM。

之后，调用 handler.processStream() 进入对来自 chaincode 容器消息处理的主循环。

## handler.processStream() 主消息循环

Peer 侧维护一个到 cc 的双向流，循环处理消息。主要在 `func (handler *Handler) processStream() error` 方法中（cc 到 peer 注册后会自动调用到该方法）。

主循环过程代码如下：

```
    for {
        in = nil
        err = nil
        nsInfo = nil
        if recv {
            recv = false
            go func() {
                var in2 *pb.ChaincodeMessage
                in2, err = handler.ChatStream.Recv()
                msgAvail <- in2
            }()
        }
        select {
        case sendErr := <-errc:
            if sendErr != nil {
                return sendErr
            }
            //send was successful, just continue
            continue
        case in = <-msgAvail:
            // Defer the deregistering of the this handler.
            if err == io.EOF {
                chaincodeLogger.Debugf("Received EOF, ending chaincode support stream,
 %s", err)
                return err
            } else if err != nil {
                chaincodeLogger.Errorf("Error handling chaincode support stream: %s",
err)
                return err
```

```
            } else if in == nil {
                err = fmt.Errorf("Received nil message, ending chaincode support strea
m")
                chaincodeLogger.Debug("Received nil message, ending chaincode support
stream")
                return err
            }
            chaincodeLogger.Debugf("[%s]Received message %s from shim", shorttxid(in.T
xid), in.Type.String())
            if in.Type.String() == pb.ChaincodeMessage_ERROR.String() {
                chaincodeLogger.Errorf("Got error: %s", string(in.Payload))
            }

            // we can spin off another Recv again
            recv = true

            if in.Type == pb.ChaincodeMessage_KEEPALIVE {
                chaincodeLogger.Debug("Received KEEPALIVE Response")
                // Received a keep alive message, we don't do anything with it for now
                // and it does not touch the state machine
                continue
            }
        case nsInfo = <-handler.nextState:
            in = nsInfo.msg
            if in == nil {
                err = fmt.Errorf("Next state nil message, ending chaincode support str
eam")
                chaincodeLogger.Debug("Next state nil message, ending chaincode suppor
t stream")
                return err
            }
            chaincodeLogger.Debugf("[%s]Move state message %s", shorttxid(in.Txid), in
.Type.String())
        case <-handler.waitForKeepaliveTimer():
            if handler.chaincodeSupport.keepalive <= 0 {
                chaincodeLogger.Errorf("Invalid select: keepalive not on (keepalive=%d
)", handler.chaincodeSupport.keepalive)
                continue
            }

            //if no error message from serialSend, KEEPALIVE happy, and don't care abo
ut error
            //(maybe it'll work later)
            handler.serialSendAsync(&pb.ChaincodeMessage{Type: pb.ChaincodeMessage_KEE
PALIVE}, nil)
            continue
        }

        err = handler.HandleMessage(in)
        if err != nil {
            chaincodeLogger.Errorf("[%s]Error handling message, ending stream: %s", sh
orttxid(in.Txid), err)
            return fmt.Errorf("Error handling message, ending stream: %s", err)
```

```
        }

        if nsInfo != nil && nsInfo.sendToCC {
            chaincodeLogger.Debugf("[%s]sending state message %s", shorttxid(in.Txid),
 in.Type.String())
            //ready messages are sent sync
            if nsInfo.sendSync {
                if in.Type.String() != pb.ChaincodeMessage_READY.String() {
                    panic(fmt.Sprintf("[%s]Sync send can only be for READY state %s\n"
, shorttxid(in.Txid), in.Type.String()))
                }
                if err = handler.serialSend(in); err != nil {
                    return fmt.Errorf("[%s]Error sending ready  message, ending stream
: %s", shorttxid(in.Txid), err)
                }
            } else {
                //if error bail in select
                handler.serialSendAsync(in, errc)
            }
        }
    }
```

首先是利用 select 结构尝试读取各种消息。包括：

- `case in = <-msgAvail` ：从 cc 侧读取到请求消息；
- `case nsInfo = <-handler.nextState` ：读取切换到下个状态的附加消息。
- `case <-handler.waitForKeepaliveTimer()` ：定期发出心跳刷新消息。

读取到合法消息后，会分别调用 `handler.HandleMessage(in)` 处理 cc 消息；以及检查状态切换消息（仅允许消息类型为 READY，意味着此时 cc 在正常运行状态），是否要发送给 cc 侧（sendToCC 为 True）。

## FSM

定义的状态、事件主要在 `func newChaincodeSupportHandler(chaincodeSupport *ChaincodeSupport, peerChatStream ccintf.ChaincodeStream) *Handler` 方法中。

一般对应 GET_STATE、GET_STATE_BY_RANGE 等简单事件，调用 handleXXX 方法。

PUT_STATE、DEL_STATE、INVOKE_CHAINCODE 三个事件，则会触发 enterBusyState() 方法。

# comm

一些简单的常见函数。

# testdata

# certs

# grpc

## impersonation

## prime256v1-openssl-cert.pem

# prime256v1-openssl-key.pem

# config.go

cache 配置中的一些变量的值。

# connection.go

跟 GRPC 连接相关的一些方法。

```
func NewClientConnectionWithAddress(peerAddress string, block bool, tslEnabled bool, c
reds credentials.TransportAuthenticator) (*grpc.ClientConn, error)
```

向指定地址建立一条 grpc 通道连接。

```
func InitTLSForPeer() credentials.TransportAuthenticator
```

返回 peer 的 TLS 客户端认证信息，从 peer.tls.cert.file 文件中读取生成。

## creds.go

## producer.go

## server.go

## util.go

# committer

Committer 角色接口和实现，实现上部分地方调用 core/peer 包中的方法。

# txvalidator

负责在 commit 阶段对区块和交易进行合法性检查。

## validator.go

主要实现 txValidator 结构体。

```
type txValidator struct {
    support Support
    vscc    vsccValidator
}
```

该结构体提供了 `func (v *txValidator) Validate(block *common.Block) error` 方法，对给定区块进行合法性检查。

主要过程如下：

- 创建一个交易过滤器，初始时给区块中所有交易都打上valid标签，然后依次检查每笔交易，再修改标签，该过滤器为了方便之后commit过程筛选有效交易。
- 对区块中的每笔交易(env)都执行 `ValidateTransaction()` , 其主要内容是检查交易的组成的完整性，签名的正确性，交易ID的正确性（检查重复交易的前提）以及交易请求Proposal的一致性（比较hash）。
- 检查交易所在通道是否存在。
- 若通道头的类型是背书交易，通过TxID检验交易是否已经存在，若否然后执行 `VSCCValidateTx()` ，最后通过获取交易instance来标记下是invoke交易还是upgrade交易。 `VSCCValidateTx()` 主要内容是获取交易读写集，检查写集的合法性（非SCC的话不能对LSCC和不可调用的SCC写入，SCC的话如果自身不能从外部调用也不可以写入），获取对应的VSCC和policy，发起对VSCC的调用来验证。
- 若通道头的类型是配置，执行 `Apply()` 去应用配置。
- 根据标记的invoke交易和upgrade交易，若有同一cc的多个upgrade交易，保留最后一个，前面的都标记为非法; 若invoke交易对应的cc（在该区块中）存在upgrade交易，则把所有该cc的invoke交易标记为非法。
- 把交易过滤器添加进block的metadata中，用于之后的commit过程。

# committer.go

定义 Committer 接口。Committer 角色必须实现这一接口。

```
type Committer interface {

    // Commit block to the ledger
    Commit(block *common.Block) error

    // CommitWithPvtData block and private data into the ledger
    CommitWithPvtData(blockAndPvtData *ledger.BlockAndPvtData) error

    // GetPvtDataAndBlockByNum retrieves block with private data with given
    // sequence number
    GetPvtDataAndBlockByNum(seqNum uint64) (*ledger.BlockAndPvtData, error)

    // Get recent block sequence number
    LedgerHeight() (uint64, error)

    // Gets blocks with sequence numbers provided in the slice
    GetBlocks(blockSeqs []uint64) []*common.Block

    // Closes committing service
    Close()
}
```

其中，最重要的是 `Commit(block *common.Block) error` 方法，负责完成对某个区块的验证的最终确认提交。

# committer_impl.go

LedgerCommitter 结构体实现了 Committer 接口。

```
type LedgerCommitter struct {
    ledger    ledger.PeerLedger
    validator txvalidator.Validator
    eventer   ConfigBlockEventer
}
```

实现的几个方法，都是通过调用成员 ledger 的方法来实现。

其中，最重要的是 `Commit(block *common.Block) error` 方法，负责完成对某个区块的验证的最终确认提交。

主要流程如下图所示。

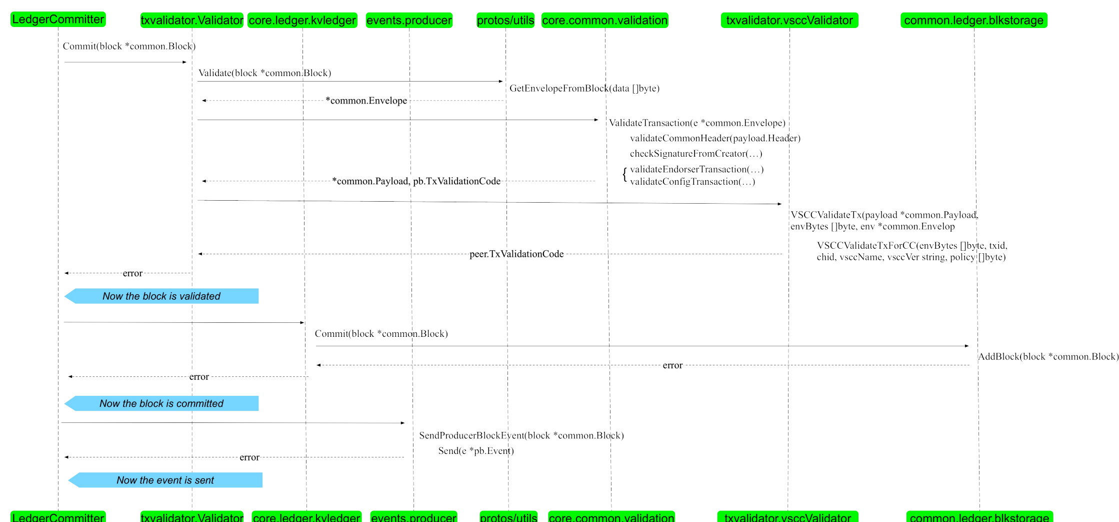



图 *1.9.4.3.1 - Commit* 流程

# common

# ccpackage

## ccpackage.go

# ccprovider

## ccinfocache.go

## ccprovider.go

## cdspackage.go

# sigcdspackage.go

# privdata 包

## collection.go

## nopcollection.go

## simplecollection.go

## store.go

# sysccprovider

# sysccprovider.go

# validation

## msgvalidation.go

# config

## config.go

# container

容器操作相关的方法实现。

chaincode 目前是运行在容器内的，这个包主要提供与之相关的操作。

# api

## core.go

# ccintf

定义 chaincode 和 peer 之间进行通信的接口，目前只有 chaincode 裸跑的时候才被使用。

# ccintf.go

比较重要的是

```
//CCID encapsulates chaincode ID
type CCID struct {
    ChaincodeSpec *pb.ChaincodeSpec
    NetworkID     string
    PeerID        string
}
```

# dockercontroller

Docker 相关的操作。

## dockercontroller.go

抽象一个 Docker 主机。

主要结构为

```
type DockerVM struct {
    id string
}
```

支持的方法包括：

- Deploy：利用给定的 tar.gz 文件生成一个镜像；
- Destroy：删除一个镜像。
- Start：启动一个 Docker 容器，命名为 网络 id + peer id + chaincode 名（hash 串），会从配置中读取 hostconfig 信息。
- Stop：停止一个 Docker 容器。

# inproccontroller

chaincode 直接裸跑在主机上时候的一些操作。

# inproccontroller.go

## inprocstream.go

# msp

## sampleconfig

# util

一些辅助方法。

## dockerutil.go

从配置中读取信息，创建一个 docker client 等。

## writer.go

将目录、包、流等的内容写到 tar 包中。

# controller.go

抽象出来的 VM 控制器，目前支持 Docker 和 系统运行，将来可以支持更多类型的容器。

# vm.go

主要数据结构为

```
type VM struct {
    Client *docker.Client
}
```

一个 VM，实际代表的是一个容器主机，目前支持列出镜像（用来测试）、生成 chaincode 容器等方法。

主要的方法是 buildChaincodeContainerUsingDockerfilePackageBytes()，根据传入的字节来生成一个 chaincode 镜像。

# deliverservice

# blocksprovider

# blocksprovider.go

# mocks

## blocksprovider.go

## orderer.go

## client.go

# deliveryclient.go

# requester.go

# endorser

Endorser 角色接口，实现上部分地方调用 peer 下面的方法。

## endorser.go

提供 Endorser 结构体。

```
// Endorser provides the Endorser service ProcessProposal
type Endorser struct {
    policyChecker policy.PolicyChecker
}
```

该结构体最关键的是提供了 `func (e *Endorser) ProcessProposal(ctx context.Context, signedProp *pb.SignedProposal) (*pb.ProposalResponse, error)` 方法，供客户端远程 grpc 调用。用来接受签名的交易提案（Signed Proposal），进行背书处理。

背书过程主要完成如下操作：

- 检查提案合法性；
- 模拟执行提案（启动链码容器，对世界状态的最新版本进行临时快照，基于它执行链码，结果记录在读写集中）；
- 对提案进行背书（对提案内容和读写集合进行签名），并返回提案响应消息。

## ProcessProposal() 方法主要过程

主要过程如下图所示。

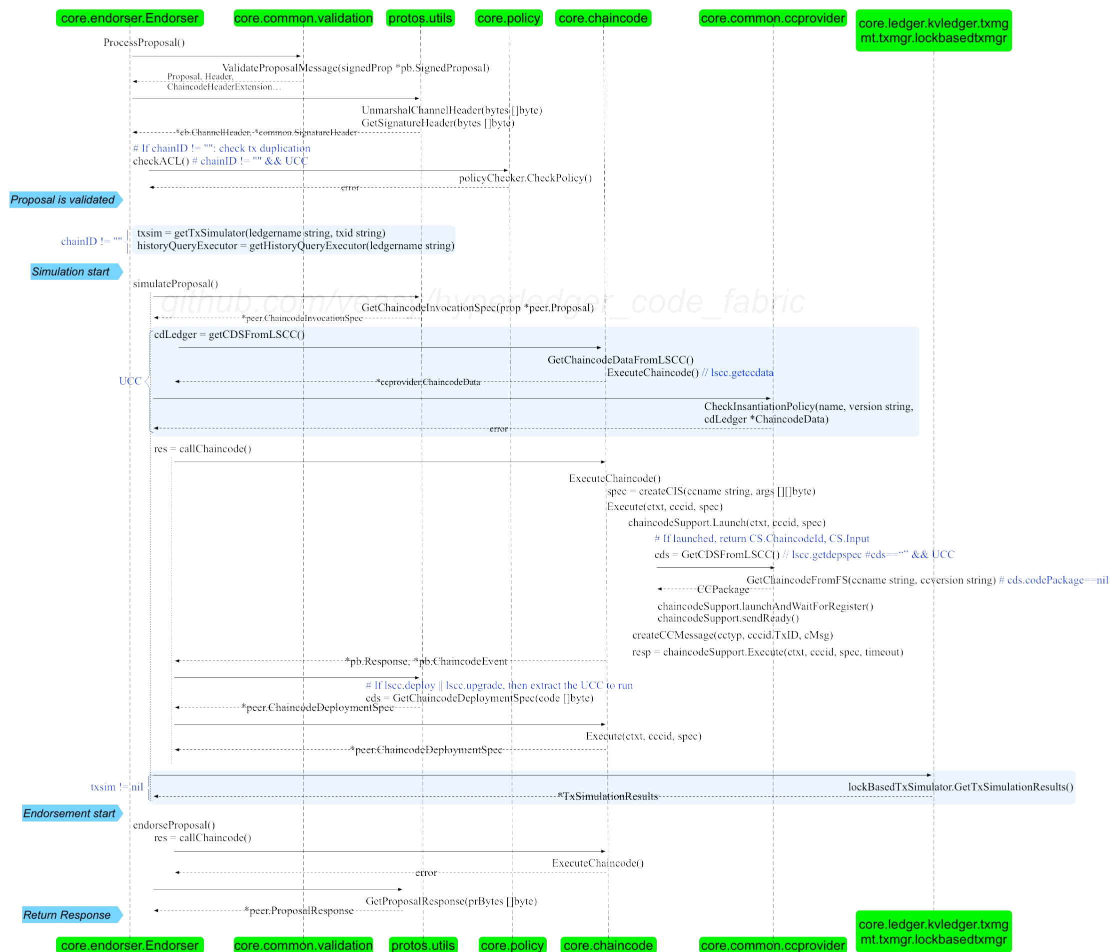



图 *1.9.9.1.1 - Endorser ProcessProposal* 过程

- 检查提案合法性；
  - 调用 ValidateProposalMessage() 方法对签名的提案进行格式检查，主要检查 Channel头（是否合法头部类型）、签名头（是否包括了 nonce和creators 数据），检查签名域（creator 是合法证书，签名是否正确）。
  - 如果是系统 CC（SCC），检查是否是允许从外部调用的三种 SCC 之一：cscc、lscc 或 qscc。
  - 如果 chainID 不为空，获取对应 chain 的账本结构，检查 TxID 的唯一性（交易在账本上没发生过）；
  - 对于用户 CC，检查 ACL。根据 chaincode 指定的 endorsement Policy，签名提案者在指定 channel 上有写权限，最终是调用 common/cauthdsl 下面代码，支持指定必须包括某个成员来签名，或者是凑够若干几个合法签名。
- 模拟执行提案
  - 如果 chainID 不为空，获取对应账本的交易模拟器（TxSimulator）和历史查询器（HistoryQueryExecutor），把 historyqueryexecutor 加入到 Context 的 K-V 储存中。

    - 如果 chainID 不为空，调用 simulateProposal() 方法获取模拟执行的结果，检查返回的响应 response的状态，若不小于错误 500 则创建并返回一个失败的 ProposalResponse。
- 对提案进行背书
    - chainID 非空情况下，调用 endorseProposal() 方法利用 ESCC，对之前得到的模拟执行的结果进行背书。返回 ProposalResponse，检查 simulateProposal 返回的 response 的状态，若不小于错误阈值 400（被背书节点反对），返回 ProposalResponse 及链码错误 chaincodeError（endorseProposal 里有检查链码执行结果的状态，而 simulateProposal 没有检查）。
    - 将 response.Payload 赋给 ProposalResponse.Response.Payload（因为 simulateProposal 返回的 response 里面包含链码调用的结果）。
    - 返回响应消息 ProposalResponse。

## simulateProposal 方法

主要过程如下：

- 从提案结构中提取 ChaincodeInvocationSpec，里面包含了所调用链码的路径、名称和版本，以及调用时传入的各种参数；
- 检查 ESCC 和 VSCC（这里其实不需要）；
- 对用户链码，检查提案中的实例化 Policy 是否跟 LSCC 得到的实例化 Policy 匹配。防止有人超出权限在其它通道实例化。
- 调用 callChaincode() 方法执行 Proposal，返回 response 和 ccevent。
    - 调用 ExecuteChaincode() 方法，该方法主要调用 Execute() 方法。调用过程中会把交易模拟器和历史查询器通过上下文结构体传入后续子方法。
    - Execute() 方法调用 theChaincodeSupport.Launch() 方法启动链码容器。Launch() 方法会创建 CC 容器并启动。
    - Execute() 方法进一步调用 theChaincodeSupport.Execute() 方法发送消息给 CC 容器，执行相关的合约。
- 对 transactionSimulator 执行 GetTxSimulationResults()拿到交易读写集 simResult。
- 返回链码数据 ChaincodeData（LSCC 中的）、响应 response、交易读写集 simResult、链码事件信息 ccevent。

## endorseProposal 方法

主要过程如下：

- 获取被调用的链码指定的背书链码的名字。
- 通过 callChaincode() 实现对背书链码的调用，返回响应 response（对 ESCC 的调用同样也会产生 simulation results，但 ESCC 不能背书自己产生的simulation results，需要背书最初被调用的链码产生的 simulation results）。

- 检查 response.Status，是否大于等于 400（错误阈值），若是则把 response 赋给 proposalResponse.Response 并返回 proposalResponse。
- 将 response.Payload解码后（ProposalResponse类型）返回。

## callChaincode 方法

主要过程如下：

- 判断交易模拟器，不为空则把它加入到Context的K-V存储中。
- 判断被call的cc是不是系统链码，创建CCContext（包含通道名、链码名、版本号、交易 ID、是否 SCC、签名 Prop、Prop）
- 调用 core/chaincode/chaincodeexec.go 下的 ExecuteChaincode()，返回响应 response 和 事件ccevent。
- 返回 response和ccevent。

# java.go

# nojava.go

# handlers 包

# auth 包

## filter 包

## plugin 包

## auth.go

## decoration 包

## decorator 包

## plugin 包

# decoration.go

# library 包

## library.go

## registry.go

# ledger

账本的实现，包括区块链（blockchain）和世界状态（world state）。

# customtx 包

## custom_tx_processor.go

# kvledger

## example

## history

## marble_example

# txmgmt

### validateTx(txRWSet, updates)

参数注释：txRWSet交易模拟结果/读写集，updates账本更新（顺序校验一个区块中的交易，每成功验证一个交易，就把交易的写集放进去）

- 该校验只校验读写集中的读集以及范围查询，不做写集的校验（不需要）。
- 针对读集，就是对于每一个Key检查是否在updates中存在，如果有，验证就不成功，另外还要检查当前账本（截至上一个区块）中它的version是否跟读集（背书节点模拟时）中的version一致，因为交易受orderer排序影响以及网络延迟影响，检查version是保证该交易之前没有交易变更过要读的Key，保证信息读取的准确性。
- 针对范围查询，就是在现有账本状态（截至上一个区块）+update中重新执行范围查询，比对结果是否还是一样。

## kv_ledger.go

实现一个支持键值对的 peer 账本结构体。

```
type kvLedger struct {
    ledgerID   string
    blockStore blkstorage.BlockStore
    txtmgmt    txmgr.TxMgr
    historyDB  historydb.HistoryDB
}
```

## kv_ledger_provider.go

## recovery.go

# ledgerconfig

## ledger_config.go

# ledgermgmt

# ledger_mgmt.go

## ledger_mgmt_test_exports.go

## ledgerstorage 包

## store.go

# pvtdatastorage 包

## kv_encoding.go

## store.go

## store_impl.go

# test_exports.go

# testutil

## test_util.go

# util

## couchdb

## txvalidationflags.go

# util.go

辅助函数。

# ledger_interface.go

定义账本结构相关的接口。

主要包括：

- HistoryQueryExecutor：负责执行历史查询，跟历史数据库打交道。
- PeerLedger：存有所有的交易的账本结构，存在于 peer 侧。跟 orderer 侧账本的区别在于带有交易合法状态的标记。
- PeerLedgerProvider：对 ledger 实例的句柄。
- QueryExecutor：负责执行对账本的查询类操作。
- TxSimulator：endorse 阶段，模拟在当前最新的世界状态上执行交易。
- ValidatedLedger：仅存在 committing 阶段，通过验证的合法交易。

# mocks

# ccprovider

## ccprovider.go

# txvalidator

## support.go

# validator

## validator.go

# peer

构成一个 fabric peer 的核心实现。

主要包括：

- peer 包：负责定义一个 peer 节点具有的各种行为。
- handler 包：处理 peer 收到的各种消息，十分关键。

# testdata

## Org1-cert.pem

## Org1-server1-cert.pem

## Org1-server1-key.pem

## Org2-cert.pem

## Org2-child1-cert.pem

## Org2-child1-key.pem

# Org2-child1-server1-cert.pem

## Org2-child1-server1-key.pem

## Org2-server1-cert.pem

## Org2-server1-key.pem

## Org3-cert.pem

## Org3-server1-cert.pem

## Org3-server1-key.pem

## generate.go

# config.go

跟配置相关的一些方法。

主要是配置的一些选项可能需要一些计算和检测，通过这些方法可以做一些 cache 等。

包括下面一些配置项：

```
var localAddress string
var localAddressError error
var peerEndpoint *pb.PeerEndpoint
var peerEndpointError error

// Cached values of commonly used configuration constants.
var syncStateSnapshotChannelSize int
var syncStateDeltasChannelSize int
var syncBlocksChannelSize int
var validatorEnabled bool
```

# peer.go

## peer 相关

比较核心的数据结构。

Peer 接口，两个方法：包括获取一个 peer 端点和发送探测 hello 消息。

```
type Peer interface {
    GetPeerEndpoint() (*pb.PeerEndpoint, error)
    NewOpenchainDiscoveryHello() (*pb.Message, error)
}
```

具体实现的数据结构为 PeerImpl。

```
type PeerImpl struct {
    handlerFactory HandlerFactory  // 生成一个 MessageHandler
    handlerMap     *handlerMap     // 所有注册上来的消息处理器
    ledgerWrapper  *ledgerWrapper  // ledger 操作句柄
    secHelper      crypto.Peer     // 处理身份验证和安全相关
    engine         Engine          // handler 工厂 + 本地交易处理的引擎
    isValidator    bool
    reconnectOnce  sync.Once
    discHelper     discovery.Discovery //探测任务句柄
    discPersist    bool
}
```

核心方法，包括：

- ExecuteTransaction：准备执行一个交易，发送给本地的引擎（VP 节点），或给远端的 peer；
- Broadcast：向所有注册的消息句柄发送消息；
- Unicast：向指定的 peer 发送消息；
- 一系列 Get 方法：包括获取 Block 内容、获取当前链的大小、当前状态的 hash、获取已注册的 peer 端点、远端 ledger 等等，很多功能实际上都是通过其它包来完成。
- sendTransactionsToLocalEngine：交易发给本地的引擎处理；
- SendTransactionsToPeer：交易发给其它 peer 处理；

## 消息相关

两个基础接口 MessageHandler 和 MessageHandlerCoordinator。

```
type MessageHandler interface {
    RemoteLedger    // 获取远端的 ledger
    HandleMessage(msg *pb.Message) error // 接收到某个消息进行处理
    SendMessage(msg *pb.Message) error //发送消息到对端
    To() (pb.PeerEndpoint, error) // 对端是哪个节点
    Stop() error
}
```

```
type MessageHandlerCoordinator interface {
    Peer
    SecurityAccessor
    BlockChainAccessor
    BlockChainModifier
    BlockChainUtil
    StateAccessor
    RegisterHandler(messageHandler MessageHandler) error
    DeregisterHandler(messageHandler MessageHandler) error
    Broadcast(*pb.Message, pb.PeerEndpoint_Type) []error
    Unicast(*pb.Message, *pb.PeerID) error
    GetPeers() (*pb.PeersMessage, error)
    GetRemoteLedger(receiver *pb.PeerID) (RemoteLedger, error)
    PeersDiscovered(*pb.PeersMessage) error
    ExecuteTransaction(transaction *pb.Transaction) *pb.Response
    Discoverer
}
```

# policy

# mocks

## mocks.go

# policy.go

# policyprovider

# provider.go

# scc

System Chaincode

已更新escc、lscc和vscc。

# cscc

## configure.go

# escc

Endorsement System Chaincode

负责对模拟执行结果进行签名。

目前只有签名功能，以后会扩展。

进一步的分析请查阅 `endorser_onevalidsignature_go.md` 。

## endorser_onevalidsignature.go

Invoke()用来背书特定的Proposal。

目前，只能对输入进行签名并返回背书的结果。以后会对chaincode进行扩展，来提供更复杂的背书策略过程。例如把策略详细信息编码成链码的调用交易，以及允许客户通过参数去选择使用哪一个背书策略。

调用ESCC时有4个必须的参数，还有两个可选。

| 序号 | 内容 | 备注 |
| --- | --- | --- |
| 0 | 函数名 | 目前未使用 |
| 1 | 序列化的Header | 必需 |
| 2 | 序列化的ChaincodeProposalPayload | 必需 |
| 3 | 被调用的链码的ChaincodeID | 必需 |
| 4 | 链码调用的结果（Response） | 必需 |
| 5 | 模拟结果（读写集） | 必需 |
| 6 | 序列化的事件 | 非必需 |
| 7 | 可见度 | 非必需，目前是完全可见 |

主要过程如下：

- 检查各个参数。
- 获取该节点的签名身份。
- 根据传进来的参数创建ProposalResponse。
- 将ProposalResponse编码为prBytes，返回shim.Success(prBytes)。

# lccc

Lifecycle System Chaincode。

负责Chaincode的安装、部署、更新、查询等操作。

进一步的分析请查阅 `lscc_go.md` 。

# lccc.go

LSCC---Lifecycle System Chaincode, 负责chaincode的全生命周期管理。

Invoke操作有INSTALL、DEPLOY(语义上的实例化)、UPGRADE、GETCCINFO、GETDEPSPEC、GETCCDATA、GETCHAINCODES、GETINSTALLEDCHAINCODES。

| 操作 | 参数 |
|---|---|
| INSTALL | 1.ChaincodeDeploymentSpec |
| DEPLOY<br>UPGRADE | 1.chainName<br>2.ChaincodeDeploymentSpec<br>3.SignaturePolicyEnvelope(endorsement policy)<br>4.escc<br>5.vscc |
| GETCCINFO<br>GETDEPSPEC<br>GETCCDATA | 1.chainName<br>2.ccName |
| GETCHAINCODES<br>GETINSTALLEDCHAINCODES | None |

### INSTALL

- 验证SignedProposal是不是admin节点签署的（check时传入的ADMINS）。
- executeInstall()
  - 检查cc名字和版本是否合法，不包含非法字符。
  - 把ccPackage写入文件系统。

# DEPLOY

- 检查参数，如果空则赋值默认值，policy默认是被任意一个成员签署，escc和vscc默认就是系统的escc和vscc，暂不支持自己指定，因为后面会检测指定的cc是不是scc，而scc目前就是固定的这几个不能动态增加。
- executeDeploy()
  - 检查cc名字和版本是否合法，不包含非法字符（以及ACL，access control，TODO）。
  - 检查链码是否已经存在，就是已经被实例化。
  - 获取ccPackage并转换成ChaincodeData。
  - 验证实例化策略，ccPackage有两种，源码打包生成的，还有install时直接传入的，直接传入的包有可能是被签名的，被签名的ccPackage是被指定了实例化策略的，只有指定的身份才可以实例化这个cc，其他所有的没被签名的ccPackage默认任意一个ADMIN节点都可以实例化。平常用的install命令，本地传入地址和参数打包生成的

ccPackage都是未被签名的。
    - createChaincode()，检查指定的escc和vscc是不是scc（所以暂不支持自己的背书和验证链码），最后putState()，其中key是cc名字，value是序列化后的ChaincodeData。

## UPGRADE

- 跟DEPLOY类似，参考DEPLOY，多了一步检查版本号是不是跟旧的版本号不同。

## GETCCINFO GETDEPSPEC GETCCDATA

- 检查通道Readers策略，是否有可读权限。
- 检查cc是否被实例化，成功则按照函数名返回相应信息。
- GETCCINFO返回cc名（成功获取ChaincodeData）。
- GETDEPSPEC返回序列化后的ChaincodeDeploymentSpec。
- GETCCDATA返回序列化后的ChaincodeData。

## GETCHAINCODES GETINSTALLEDCHAINCODES

- 检查是不是ADMIN节点。
- GETCHAINCODES返回所有已经实例化的cc（对LSCC的state进行范围查询）。
- GETINSTALLEDCHAINCODES返回节点上所有安装的cc（不一定实例化）。

# qscc

## query.go

# rscc 包

## rscc.go

# rsccpolicy.go

# samplesyscc

## samplesyscc.go

# vscc

Validate System Chaincode

负责背书的校验。

相对于目前escc只做签名，vscc的工作也就是验证（每个）签名的有效性，以及是否符合背书策略（有效签名个数是否满足）。

特殊地，vscc中有一部分专门针对lscc的校验。

进一步的分析请查阅 `validator_onevalidsignature_go.md` 。

## validator_onevalidsignature.go

参数

| 序号 | 内容 | 备注 |
|---|---|---|
| 1 | 函数名 | 未使用 |
| 2 | 序列化的Env | 无 |
| 3 | 序列化的policy | 无 |

**Invoke()**

- 检查参数。
- 获取policy，`NewPolicy()`会把传入的签名策略编译成验证函数。签名策略有两种类型，一种是NOutOf（就是我们看到的AND和OR组合的背书策略，本质上是多个NOutOf的组合，表达一定个数中至少要有几个的意思），一种是SignedBy（由特定的身份签名），对于编译出来的验证函数，SignedBy就是校验给定的SignedData数组中有没有该特定身份签名的，有就返回TRUE，而NOutOf相当于SignedBy的数组，根据里面所有SignedBy返回的TRUE的个数是否满足设定的数值来返回TRUE或FAUSE。
- 从交易中抽取出所有的背书endorsements，去掉重复的身份identity（背书节点），构建一个签名集signatureSet，实质上是一个SignedData的数组，SignedData由原始数据、身份和签名组成。
- 执行`policy.Evaluate(signatureSet)`，就是去执行编译出来的验证函数，其验证思路参考上面获取policy步骤。
- 另外如果被调用的cc是LSCC的话，则执行特殊的validate过程，`ValidateLSCCInvocation()`，主要检查调用LSCC的参数，如果是实例化或更新操作则另外检查cc是否install，写集是否有且只有两个（一个LSCC，一个cc），且符合LSCC写入的规范（Key必须是要实例化或更新的cc的名字，Value必须是ChaincodeData且名字版本都匹配，对LSCC只能putstate()一次），最后检查instantiatePolicy。
- 返回shim.Success(nil)。

# importsysccs.go

# loadsysccs.go

# sccproviderimpl.go

# sysccapi.go

# testutil

# config.go

# transientstore 包

# store.go

# store_helper.go

# test_exports.go

# admin.go

对 peer 服务的管理：启动和停止、查看等。

# fsm.go

peer connection 的状态机。

```
func NewPeerConnectionFSM(to string) *PeerConnectionFSM {
    d := &PeerConnectionFSM{
        To: to,
    }

    d.FSM = fsm.NewFSM(
        "created",
        fsm.Events{
            {Name: "HELLO", Src: []string{"created"}, Dst: "established"},
            {Name: "GET_PEERS", Src: []string{"established"}, Dst: "established"},
            {Name: "PEERS", Src: []string{"established"}, Dst: "established"},
            {Name: "PING", Src: []string{"established"}, Dst: "established"},
            {Name: "DISCONNECT", Src: []string{"created", "established"}, Dst: "closed"
},
        },
        fsm.Callbacks{
            "enter_state":  func(e *fsm.Event) { d.enterState(e) },
            "before_HELLO": func(e *fsm.Event) { d.beforeHello(e) },
            "after_HELLO":  func(e *fsm.Event) { d.afterHello(e) },
            "before_PING":  func(e *fsm.Event) { d.beforePing(e) },
            "after_PING":   func(e *fsm.Event) { d.afterPing(e) },
        },
    )

    return d
}
```

# devenv

主要是方便本地搭建开发平台的一些脚本。

# images

# tools

# couchdb

# Vagrantfile

# failure-motd.in

# golang_buildcmd.sh

# golang_buildpkg.sh

# install_nvm.sh

# limits.conf

# setup.sh

# setupRHELonZ.sh

# setupUbuntuOnPPC64le.sh

# docs

项目相关的所有文档。

# custom_theme

# searchbox.html

# source

# Gerrit

# Style-guides

# _static

# css

# _templates

## footer.html

## layout.html

# dev-setup

## headers.txt

## images

# DCO1.1.txt

# conf.py

# mdtorst.sh

# requirements.txt

# Makefile

# requirements.txt

# events

EventHub 服务处理相关的模块。

Event 包括四种类型：

```
enum EventType {
        REGISTER = 0;
        BLOCK = 1;
    CHAINCODE = 2;
    REJECTION = 3;
}
```

# consumer

事件消费者，负责从系统中获取事件。

客户端如果想监听系统中事件，可以通过使用消费者模块提供的方法进行。

主要代码在 consumer.go。

# adapter.go

定义一个事件的 Adapter 接口，获取感兴趣的事件类型，接收事件。

```
//EventAdapter is the interface by which a fabric event client registers interested ev
ents and
//receives messages from the fabric event Server
type EventAdapter interface {
    GetInterestedEvents() ([]*peer.Interest, error)
    Recv(msg *peer.Event) (bool, error)
    Disconnected(err error)
}
```

# consumer.go

消费者模块的主要结构和实现。

主要实现了 EventsClient 结构体，供用户使用获取系统中事件。

```
type EventsClient struct {
    sync.RWMutex
    peerAddress string
    regTimeout  time.Duration
    stream      ehpb.Events_ChatClient
    adapter     EventAdapter
}
```

提供几个主要方法：

- `func (ec *EventsClient) Start() error` ：入口方法，该方法会自动创建到 peer 事件流的连接，并开始处理收到的消息。
- `func (ec *EventsClient) Recv() (*ehpb.Event, error)` ：从流连接中接收一个事件。
- `func (ec *EventsClient) Stop() error` ：终止事件流连接。

启动后主要调用过程如下图所示。

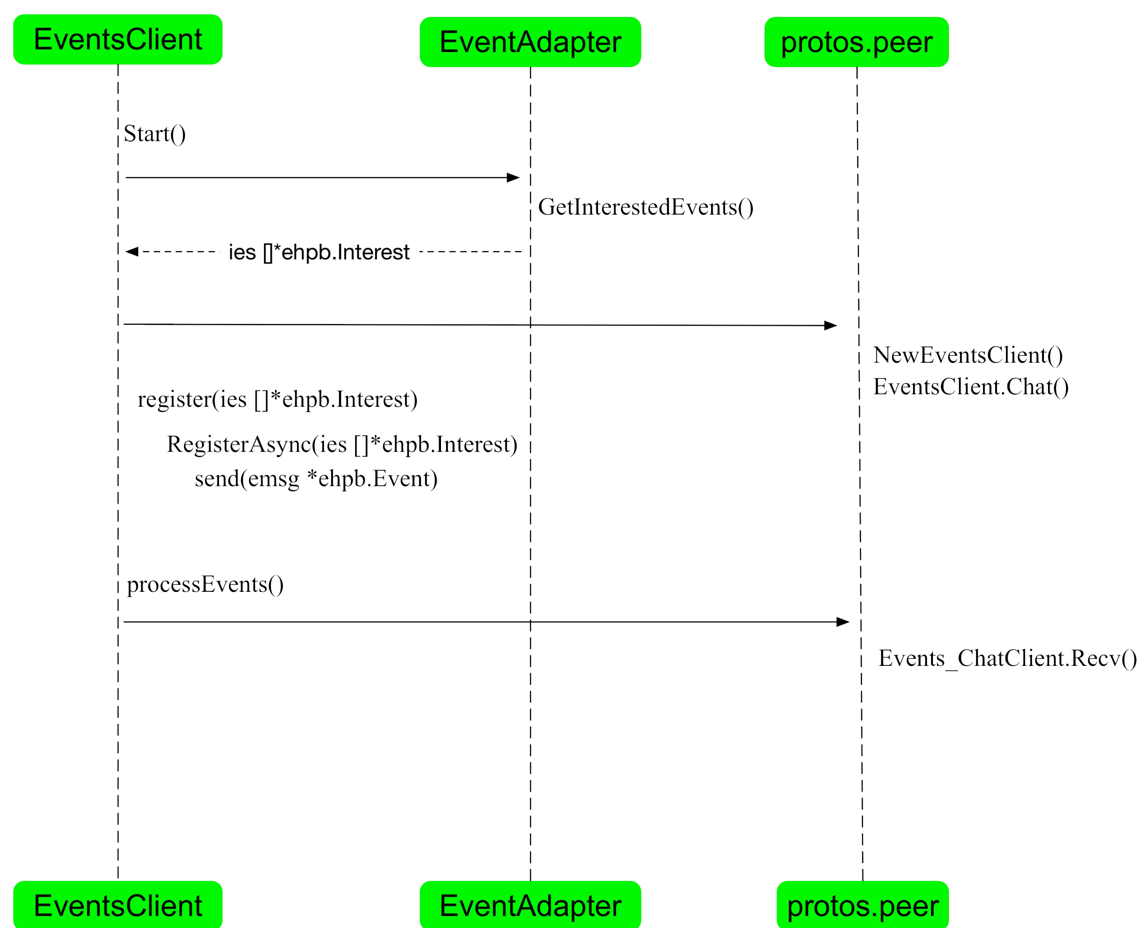



图 *1.12.1.2.1 - consumer 客户端启动后流程*

# producer

事件产生者，负责提供一个事件服务器。

任何希望生成事件的模块，可以使用产生模块提供的方法进行。

主要代码在 producer.go。

# eventhelper.go

# events.go

事件产生的入口。

主要实现了 eventProcessor 结构体，负责实现对事件的处理。

```
type eventProcessor struct {
    sync.RWMutex
    eventConsumers map[pb.EventType]handlerList

    //we could generalize this with mutiple channels each with its own size
    eventChannel chan *pb.Event

    //timeout duration for producer to send an event.
    //if < 0, if buffer full, unblocks immediately and not send
    //if 0, if buffer full, will block and guarantee the event will be sent out
    //if > 0, if buffer full, blocks till timeout
    timeout time.Duration
}
```

start() 方法启动后会进入主循环，不断检测 eventChannel 中是否有事件（由组件产生），如果有，则调用对应的 handler 处理。

# handler.go

handler 是针对某种消息的处理句柄。

```
type handler struct {
    ChatStream       pb.Events_ChatServer
    interestedEvents map[string]*pb.Interest
}
```

# producer.go

## NewEventsServer() 方法

该方法会创建一个全局的事件服务器（EventsServer），并完成初始化工作。

```
// NewEventsServer returns a EventsServer
func NewEventsServer(bufferSize uint, timeout time.Duration) *EventsServer {
    if globalEventsServer != nil {
        panic("Cannot create multiple event hub servers")
    }
    globalEventsServer = new(EventsServer)
    initializeEvents(bufferSize, timeout)
    //initializeCCEventProcessor(bufferSize, timeout)
    return globalEventsServer
}
```

代码中会初始化两个全局变量：

- globalEventsServer：全局唯一的事件服务器。
- gEventProcessor：全局唯一的 事件处理器。

initializeEvents(bufferSize, timeout)方法会注册内部消息类型（包括 EventType_BLOCK、EventType_CHAINCODE、EventType_REJECTION 和 EventType_REGISTER），并启动 gEventProcessor。

```
func initializeEvents(bufferSize uint, tout time.Duration) {
    if gEventProcessor != nil {
        panic("should not be called twice")
    }

    gEventProcessor = &eventProcessor{eventConsumers: make(map[pb.EventType]handlerList), eventChannel: make(chan *pb.Event, bufferSize), timeout: tout}

    addInternalEventTypes()

    //start the event processor
    go gEventProcessor.start()
}
```

gEventProcessor 启动后会不断从 eventChannel 通道中读取消息，并调用绑定的 handler 的 SendMessage() 方法发送出去。

## EventsServer 结构体

主要实现了事件服务器结构体。

```
// EventsServer implementation of the Peer service
type EventsServer struct {
}
```

结构体实现了如下方法：

- `func (p *EventsServer) Chat(stream pb.Events_ChatServer) error`：调用该方法会发送消息到该事件服务器。事件服务器会生成一个新的事件处理句柄，并调用句柄的 HandleMessage() 方法对消息进行处理。

# register_internal_events.go

# examples

示例文件，包括一些 chaincode 示例和监听事件的示例。

# ccchecker

# chaincodes

## newkeyperinvoke

## chaincodes.go

## registershadow.go

# ccchecker.go

# ccchecker.json

# init.go

## main.go

# chaincode

# chaintool

## example02

# go

## chaincode_example01

## chaincode_example02

## chaincode_example03

## chaincode_example04

## chaincode_example05

# enccc_example 包

## eventsender

## invokereturnsvalue

## map

## marbles02

## passthru

## sleeper

## utxo

# java

## Example

## LinkExample

## MapExample

## RangeExample

## SimpleSample

## chaincode_example02

## chaincode_example04

## chaincode_example05

## chaincode_example06

# eventsender

# cluster

# compose

## compose-up.sh.in

## configure.sh.in

## docker-compose.yaml.in

# config

## configtx.yaml

## core.yaml

## cryptogen.yaml

## fabric-ca-server-config.yaml

## fabric-tlsca-server-config.yaml

## orderer.yaml

# Makefile

# usage.txt

# configtxupdate

# bootstrap_batchsize

## script.sh

# common_scripts

## common.sh

# reconfig_batchsize

## script.sh

# reconfig_membership

## script.sh

# e2e_cli

# base

## docker-compose-base.yaml

## peer-base.yaml

# channel-artifacts

# crypto-config

## ordererOrganizations

## peerOrganizations

## examples

## chaincode

# scripts

## script.sh

# configtx.yaml

# crypto-config.yaml

## docker-compose-cli.yaml

# docker-compose-couch.yaml

# docker-compose-e2e-template.yaml

# docker-compose-e2e.yaml

# download-dockerimages.sh

# generateArtifacts.sh

# network_setup.sh

# events

# block-listener

## block-listener.go

# experimental 包

# experimental.go

# interface.go

# interface_stable.go

# stable.go

# plugins 包

# bccsp 包

## plugin.go

# scc 包

## plugin.go

# gossip

# api

# channel.go

# crypto.go

# subchannel.go

# comm

## mock

## mock_comm.go

## ack.go

# comm.go

# comm_impl.go

# conn.go

# crypto.go

# demux.go

# msg.go

# common

# common.go

## metastate.go

# discovery

# discovery.go

# discovery_impl.go

# election

# adapter.go

# election.go

# filter

# filter.go

# gossip

# algo

## pull.go

## channel

## channel.go

## msgstore

## msgs.go

# pull

## pullstore.go

# batcher.go

# certstore.go

# chanstate.go

# gossip.go

# gossip_impl.go

# identity

# identity.go

# integration

# integration.go

# privdata 包

# coordinator.go

# dataretriever.go

# distributor.go

# pull.go

## util.go

# service

# eventer.go

# gossip_service.go

# state

# mocks

## gossip.go

# payloads_buffer.go

## state.go

# util

## logging.go

## misc.go

# msgs.go

# privdata.go

# pubsub.go

# gotools

go 相关的开发工具的安装脚本：golint、govendor、goimports、protoc-gen-go、ginkgo、gocov、gocov-xml 等。

- golint：支持 golang 的静态语法和格式检查；
- govendor：管理第三方引入包；
- goimports：格式化 import 的包；
- protoc-gen-go：对 protobuf 的支持；
- ginkgo：支持 BDD 的框架；
- gocov：golang 的单元测试覆盖率检查工具；
- gocov-xml：支持 gocov 生成 xml 格式的报告。

在该包下执行 `make` 或在上层执行 make gotools 会自动安装上述工具。

# Makefile

# idemix 包

# credential.go

# credrequest.go

# idemix.pb.go

# issuerkey.go

# nymsignature.go

# signature.go

# util.go

# images

一些跟 Docker 镜像生成相关的配置和脚本。主要包括各个镜像的 Dockerfile.in 文件。这些文件是生成 Dockerfile 的模板。

主要包括如下镜像的 Dockerfile 的模板和生成相关脚本：

- ccenv：golang chaincode 运行的基础环境，包括 chaintool、protoc-gen-go 和 goshim 包。
- javaenv：运行 java chaincode 的基础环境，包括 javashim、protos、maven、java 环境等。
- kafka：kafka 镜像。
- order：fabric-orderer 镜像，包括 orderer 二进制文件、orderer.yaml 配置文件、msp-sampleconfig 包等。
- peer：fabric-peer 镜像，包括 peer 二进制文件、core.yaml 配置文件、msp-sampleconfig、genesis-sampleconfig 包等。
- tetenv：测试环境镜像，包括 gotools、peer、orderer 等二进制和配置文件、msp-sampleconfig、softhsm2。
- zookeeper：zookeeper 镜像。

# ccenv

生成 chaincode 运行的环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
COPY payload/chaintool payload/protoc-gen-go /usr/local/bin/
ADD payload/goshim.tar.bz2 $GOPATH/src/
```

## Dockerfile.in

# couchdb

# Dockerfile.in

# docker-entrypoint.sh

# local.ini

# vm.args

# javaenv

生成 java chaincode 运行的环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
RUN wget https://services.gradle.org/distributions/gradle-2.12-bin.zip -P /tmp --quiet
RUN unzip -q /tmp/gradle-2.12-bin.zip -d /opt && rm /tmp/gradle-2.12-bin.zip
RUN ln -s /opt/gradle-2.12/bin/gradle /usr/bin
ADD payload/javashim.tar.bz2 /root
ADD payload/protos.tar.bz2 /root
ADD payload/settings.gradle /root
WORKDIR /root
# Build java shim after copying proto files from fabric/proto
RUN core/chaincode/shim/java/javabuild.sh
```

## Dockerfile.in

# kafka

# Dockerfile.in

# docker-entrypoint.sh

# kafka-run-class.sh

# orderer

生成 fabric-order 镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-runtime:_TAG_
ENV ORDERER_CFG_PATH /etc/hyperledger/fabric
RUN mkdir -p /var/hyperledger/db /etc/hyperledger/fabric
COPY payload/orderer /usr/local/bin
COPY payload/orderer.yaml $ORDERER_CFG_PATH
EXPOSE 7050
CMD orderer
```

## Dockerfile.in

# peer

生成 fabric-peer 镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-runtime:_TAG_
ENV PEER_CFG_PATH /etc/hyperledger/fabric
RUN mkdir -p /var/hyperledger/db $PEER_CFG_PATH/msp/sampleconfig/signcerts $PEER_CFG_PATH/msp/sampleconfig/admincerts $PEER_CFG_PATH/msp/sampleconfig/keystore $PEER_CFG_PATH/msp/sampleconfig/cacerts
COPY payload/peer /usr/local/bin
COPY payload/core.yaml $PEER_CFG_PATH
COPY payload/msp/sampleconfig/signcerts/peer.pem $PEER_CFG_PATH/msp/sampleconfig/signcerts
COPY payload/msp/sampleconfig/admincerts/admincert.pem $PEER_CFG_PATH/msp/sampleconfig/admincerts
COPY payload/msp/sampleconfig/keystore/key.pem $PEER_CFG_PATH/msp/sampleconfig/keystore
COPY payload/msp/sampleconfig/cacerts/cacert.pem $PEER_CFG_PATH/msp/sampleconfig/cacerts
CMD peer node start
```

## Dockerfile.in

# testenv

生成一个测试环境镜像。

包括 Dockerfile.in 文件，内容为：

```
FROM hyperledger/fabric-baseimage:_BASE_TAG_
ADD payload/gotools.tar.bz2 /usr/local/bin/
WORKDIR /opt/gopath/src/github.com/hyperledger/fabric
```

## Dockerfile.in

## install-softhsm2.sh

# tools

## Dockerfile.in

# zookeeper

# Dockerfile.in

# docker-entrypoint.sh

# msp

成员服务提供者（Member Service Provider），提供一组认证相关的密码学机制和协议，用来负责对网络提供证书分发、校验，身份认证管理等。

通常情况下，一个组织可以作为一个 MSP，负责对旗下所有成员的管理。

该包下面的 `sampleconfig` 目录中提供了样例配置文件，主要包括各个证书。

# cache 包

# cache.go

# mgmt

# testtools

## config.go

# deserializer.go

## mgmt.go

# principal.go

# mocks 包

# mocks.go

# testdata

# badadmin

## admincerts

## cacerts

## keystore

## signcerts

## config.yaml

# badconfigou

## admincerts

## cacerts

## keystore

## signcerts

## config.yaml

# badconfigoucert

## admincerts

## cacerts

## keystore

## signcerts

## config.yaml

## expiration 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

# external

## admincerts

## cacerts

## intermediatecerts

## keystore

## signcerts

## config.yaml

# idemix 包

## MSP1OU1 包

## MSP1OU1Admin 包

## MSP1OU2 包

## MSP1Verifier 包

## MSP2OU1 包

## intermediate

## admincerts

## cacerts

## intermediatecerts

# keystore

## signcerts

# intermediate2

## admincerts

## cacerts

## intermediatecerts

## keystore

## signcerts

## users

# mspid 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

### tlscacerts 包

## nodeous1 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 🗀

## tlscacerts 包

## config.yaml

## nodeous2 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

## tlscacerts 包

## config.yaml

## nodeous3 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

## tlscacerts 包

## config.yaml

## nodeous4 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

## tlscacerts 包

## config.yaml

## nodeous6 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

## tlscacerts 包

## config.yaml

## nodeous7 包

## admincerts 包

## cacerts 包

## keystore 包

## signcerts 包

## tlscacerts 包

## config.yaml

## revocation

## admincerts

## cacerts

## crls

## keystore

## signcerts

# revocation2

## admincerts

## cacerts

## crls

## keystore

## signcerts

# revokedica

## admincerts

## cacerts

## crls

## intermediatecerts

## keystore

## signcerts

# tls

## admincerts

## cacerts

## intermediatecerts

## keystore

## signcerts

## tlscacerts

## tlsintermediatecerts

## config.yaml

# cert.go

# configbuilder.go

# factory.go

# idemixmsp.go

# identities.go

定义了 identity 和 signingidentity 私有结构。

```
type identity struct {
    // id contains the identifier (MSPID and identity identifier) for this instance
    id *IdentityIdentifier

    // cert contains the x.509 certificate that signs the public key of this instance
    cert *x509.Certificate

    // this is the public key of this instance
    pk bccsp.Key

    // reference to the MSP that "owns" this identity
    msp *bccspmsp
}

type signingidentity struct {
    // we embed everything from a base identity
    identity

    // signer corresponds to the object that can produce signatures from this identity
    signer *signer.CryptoSigner
}
```

# msp.go

定义了一些基础功能接口，包括

- MSPManager：管理 MSP
- MSP：服务提供者的抽象，提供签名、校验等功能。
- Identity：跟证书相关的操作，包括获取内容，校验内容等。
- SigningIdentity：继承自 Identity，进一步支持签名功能。

# mspimpl.go

对 MSP 接口的实现，实现了 bccspmsp 结构，可以通过 NewBccspMsp() 方法生成。

bccspmsp 提供一个默认的 MSP 实现，成员包括：

- Type 为 `FABRIC = 0`；
- bccsp 为 SHA-2 256；

方法主要包括：

- Setup()：利用给定的配置信息，进行初始化操作。
- GetType()：返回类型，目前为 `FABRIC`（值为 0）。
- GetIdentifier()：返回 msp 的名称。
- GetDefaultSigningIdentity()：获取本 MSP 中的默认签名个体。
- GetSigningIdentity()：获取本 MSP 中的签名个体。
- Validate（）：对给定的 Identity 对象进行校验。
- DeserializeIdentity()：从序列化对象中解析 Identity 对象。
- SatisfiesPrincipal()：检查某个 Identity 是否符合给定的策略。

- getIdentityFromConf()：从本地配置文件中解析出 x.509 格式的证书信息，包括公钥等，利用这些信息生成 Identity 对象。

- getSigningIdentityFromConf()：从本地配置文件中解析出 x.509 格式的证书信息，包括公钥等，利用这些信息生成带签名功能的 Identity 对象。

# mspimplsetup.go

# mspimplvalidate.go

# mspmgrimpl.go

# orderer

在 fabric 1.0 架构中，共识功能被抽取出来，作为单独的 fabric-orderer 模块来实现，完成核心的排序功能。最核心的功能是实现从客户端过来的 broadcast 请求，和从 orderer 发送到 peer 节点的 deliver 接口。同时，orderer 需要支持多 channel 的维护。

目前，orderer 模块支持三种排序类型：

- Solo：单节点的排序功能，试验性质，不具备可扩展性和容错；
- Kafka：基于 Kafka 集群的排序实现，支持可持久化和可扩展性；
- BFT：支持 BFT 容错的排序实现，尚未完成。

账本记录上支持两种类型：

- Ram：存放近期若干（默认为 1000 个）区块到内存中。
- File：存放区块记录到文件系统，默认是临时目录下的 `hyperledger-fabric-ordererledger` 文件。

sampleconfig 目录下有示例的配置文件 orderer.yaml。

# common

通用函数。

# blockcutter

## blockcutter.go

# bootstrap

# file

# bootstrap.go

# broadcast

负责响应 Broadcast 请求。

# broadcast.go

对 orderer 服务的 Broadcast 请求，会交给 `orderer.common.server` 包中 server 结构体的 `Broadcast(srv ab.AtomicBroadcast_BroadcastServer) error` 方法处理。该方法主要会调用到 `orderer.common.broadcast` 包中的 handlerImpl 结构的 `func (bh *handlerImpl) Handle(srv ab.AtomicBroadcast_BroadcastServer) error` 方法。

handlerImpl 结构体十分重要，完成对 Broadcast 请求的核心处理过程。

```
type handlerImpl struct {
    sm ChannelSupportRegistrar
}

func (bh *handlerImpl) Handle(srv ab.AtomicBroadcast_BroadcastServer) error
```

`Handle(srv ab.AtomicBroadcast_BroadcastServer) error` 方法会开启一个循环来从 srv 中读取请求消息并进行处理，直到结束。

解析消息

首先，解析消息，获取消息头、是否为配置消息、获取对应处理器结构。

```
chdr, isConfig, processor, err := bh.sm.BroadcastChannelSupport(msg)
```

实际上，映射到 orderer.common.server 包中 broadcastSupport 结构体的 `BroadcastChannelSupport(msg *cb.Envelope) (*cb.ChannelHeader, bool, broadcast.ChannelSupport, error)` 方法，进一步调用到 orderer.common.multichannel 包中 Registrar 结构体的对应方法。

```
func (r *Registrar) BroadcastChannelSupport(msg *cb.Envelope) (*cb.ChannelHeader, bool
, *ChainSupport, error) {
    chdr, err := utils.ChannelHeader(msg)
    if err != nil {
        return nil, false, nil, fmt.Errorf("could not determine channel ID: %s", err)
    }

    cs, ok := r.chains[chdr.ChannelId] //应用通道、系统通道
    if !ok {
        cs = r.systemChannel
    }

    isConfig := false
    switch cs.ClassifyMsg(chdr) {
    case msgprocessor.ConfigUpdateMsg:
        isConfig = true
    default:
    }

    return chdr, isConfig, cs, nil
}
```

channel 头部从消息信封结构中解析出来；是否为配置信息根据消息头中类型进行判断（是否为 cb.HeaderType_CONFIG_UPDATE）;通过字典查到对应的 ChainSupport 结构（应用通道、系统通道）作为处理器。

之后，利用解析后的结果，分别对不同类型的消息（普通消息、配置消息）进行不同处理。下面以应用通道的 ChainSupport 结构作为处理器进行介绍。

处理非配置消息

对于非配置消息，主要执行如下两个操作：消息检查和入队列操作。

```
configSeq, err := processor.ProcessNormalMsg(msg) //消息检查
processor.Order(msg, configSeq) //入队列操作
```

消息检查方法会映射到 orderer.common.msgprocessor 包中 StandardChannel 结构体的 `ProcessNormalMsg(env *cb.Envelope) (configSeq uint64, err error)` 方法，实现如下。

```
func (s *StandardChannel) ProcessNormalMsg(env *cb.Envelope) (configSeq uint64, err er
ror) {
    configSeq = s.support.Sequence() // 获取配置的序列号，映射到 common.configtx 包中 confi
gManager 结构体的对应方法
    err = s.filters.Apply(env) // 进行过滤检查，实现为 orderer.common.msgprocessor 包中 Ru
leSet 结构体的对应方法。
    return
}
```

其中，过滤器会在创建 ChainSupport 结构时候初始化：

- 应用通道：orderer.common.mspprocessor 包中的 `CreateStandardChannelFilters(filterSupport channelconfig.Resources) *RuleSet` 方法，包括 EmptyRejectRule、SizeFilter 和 SigFilter（ChannelWriters 角色）。
- 系统通道：orderer.common.mspprocessor 包中的 `CreateSystemChannelFilters(chainCreator ChainCreator, ledgerResources channelconfig.Resources) *RuleSet` 方法，包括 EmptyRejectRule、SizeFilter、SigFilter（ChannelWriters 角色）和 SystemChannelFilter。

入队列操作会根据 consensus 配置的不同映射到 orderer.consensus.solo 包或 orderer.consensus.kafka 包中的方法。

以 kafka 情况为例，会映射到 chainImpl 结构体的对应方法。该方法会将消息进一步封装为 `sarama.ProducerMessage` 类型消息，通过 enqueue 方法发给 Kafka 后端。

```
func (chain *chainImpl) Order(env *cb.Envelope, configSeq uint64) error {
    marshaledEnv, err := utils.Marshal(env)
    if err != nil {
        return fmt.Errorf("cannot enqueue, unable to marshal envelope because = %s", e
rr)
    }
    if !chain.enqueue(newNormalMessage(marshaledEnv, configSeq)) {
        return fmt.Errorf("cannot enqueue")
    }
    return nil
}
```

处理配置消息

对于配置消息，处理过程与正常消息略有不同，包括合并配置更新消息和入队列操作。

```
config, configSeq, err := processor.ProcessConfigUpdateMsg(msg) // 合并配置更新消息
processor.Configure(config, configSeq) //入队列操作
```

合并配置更新消息方法会映射到 orderer.common.msgprocessor 包中 StandardChannel 结构体的 `ProcessConfigUpdateMsg(env *cb.Envelope) (configSeq uint64, err error)` 方法，计算合并后的配置和配置编号，实现如下。

```
func (s *StandardChannel) ProcessConfigUpdateMsg(env *cb.Envelope) (config *cb.Envelop
e, configSeq uint64, err error) {
    logger.Debugf("Processing config update message for channel %s", s.support.ChainID
())

    seq := s.support.Sequence()
    err = s.filters.Apply(env)
    if err != nil {
        return nil, 0, err
    }

    configEnvelope, err := s.support.ProposeConfigUpdate(env)
    if err != nil {
        return nil, 0, err
    }

    config, err = utils.CreateSignedEnvelope(cb.HeaderType_CONFIG, s.support.ChainID()
, s.support.Signer(), configEnvelope, msgVersion, epoch)
    if err != nil {
        return nil, 0, err
    }

    err = s.filters.Apply(config)
    if err != nil {
        return nil, 0, err
    }

    return config, seq, nil
}
```

入队列操作会根据 consensus 配置的不同映射到 orderer.consensus.solo 包或 orderer.consensus.kafka 包中的方法。

以 kafka 情况为例，会映射到 chainImpl 结构体的对应方法。该方法会将消息进一步封装为 `sarama.ProducerMessage` 类型消息，通过 enqueue 方法发给 Kafka 后端。

```
func (chain *chainImpl) Configure(config *cb.Envelope, configSeq uint64) error {
    marshaledConfig, err := utils.Marshal(config)
    if err != nil {
        return fmt.Errorf("cannot enqueue, unable to marshal config because = %s", err
)
    }
    if !chain.enqueue(newConfigMessage(marshaledConfig, configSeq)) {
        return fmt.Errorf("cannot enqueue")
    }
    return nil
}
```

返回响应

如果处理成功，则返回成功响应消息。

```
err = srv.Send(&ab.BroadcastResponse{Status: cb.Status_SUCCESS})
```

# deliver

# deliver.go

对 orderer 服务的 Deliver 请求（如获取某个区块），最终会调用到这里的 deliverServer 结构的 `Handle(srv ab.AtomicBroadcast_DeliverServer) error` 方法。

```
type deliverServer struct {
    sm SupportManager
}

func (ds *deliverServer) Handle(srv ab.AtomicBroadcast_DeliverServer) error
```

Handle 方法会开启一个循环来处理请求消息，对每一个消息，调用 `deliverBlocks(srv ab.AtomicBroadcast_DeliverServer, envelope *cb.Envelope) error` 方法进行处理。

获取通道头

```
payload, err := utils.UnmarshalPayload(envelope.Payload)
chdr, err := utils.UnmarshalChannelHeader(payload.Header.ChannelHeader)
```

获取对应的链结构

```
chain, ok := ds.sm.GetChain(chdr.ChannelId)
```

检查权限

检查请求方是否具备 ChannelReaders 角色的权限。

```
sf := msgprocessor.NewSigFilter(policies.ChannelReaders, chain.PolicyManager())
if err := sf.Apply(envelope); err != nil {
    logger.Warningf("[channel: %s] Received unauthorized deliver request from %s: %s",
 chdr.ChannelId, addr, err)
    return sendStatusReply(srv, cb.Status_FORBIDDEN)
}
```

查找区块

将请求信封结构的 Payload.Data 域，转换为 `ab.SeekInfo` 类型的结构。

根据 `ab.SeekInfo` 结构中指定的 Start、Stop 范围循环发送出区块答复消息给请求方。

```
block, status := cursor.Next()
sendBlockReply(srv, block)
```

# ledger 包

各种类型的账本结构，包括ram、json 和文件等。

## file 包

## json 包

## ram 包

## ledger.go

## util.go

# localconfig 包

## config.go

## metadata 包

## metadata.go

# msgprocessor 包

## filter.go

## msgprocessor.go

## sigfilter.go

## sizefilter.go

## standardchannel.go

## systemchannel.go

## systemchannelfilter.go

## multichannel 包

## blockwriter.go

## chainsupport.go

## registrar.go

## performance 包

## server.go

## utils.go

# server 包

## testdata 包

## docker-compose.yml

## main.go

orderer 启动后的 main 方法会调用到这里的 `Main()` 方法。

核心代码非常简单。

```go
// Main is the entry point of orderer process
func Main() {
    fullCmd := kingpin.MustParse(app.Parse(os.Args[1:]))

    // "version" command
    if fullCmd == version.FullCommand() {
        fmt.Println(metadata.GetVersionInfo())
        return
    }

    conf := config.Load()
    initializeLoggingLevel(conf)
    initializeLocalMsp(conf)

    Start(fullCmd, conf)
}
```

整体的调用流程如下图所示。

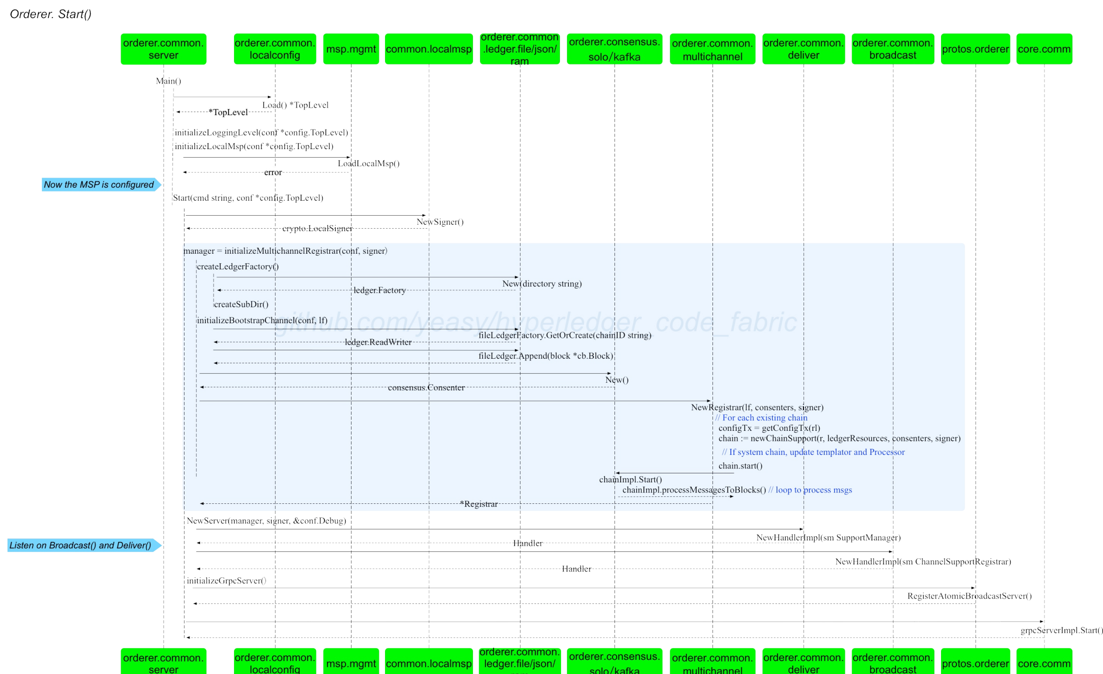



图 *1.19.1.11.3.1 - orderer.common.server* 包中的 *Main()* 方法

其中：

- config.Load()：从本地配置文件和环境变量中读取配置信息，构建配置树结构。
- initializeLoggingLevel(conf)：配置日志级别。
- initializeLocalMsp(conf)：配置 MSP 结构。
- Start()：完成启动后的核心工作。

### initializeLocalMsp

根据 orderer.yaml 配置中路径，读取本地的 msp 数据。

```
func initializeLocalMsp(conf *config.TopLevel) {
    // Load local MSP
    err := mspmgmt.LoadLocalMsp(conf.General.LocalMSPDir, conf.General.BCCSP, conf.General.LocalMSPID)
    if err != nil { // Handle errors reading the config file
        logger.Fatal("Failed to initialize local MSP:", err)
    }
}
```

### Start()

Start() 完成了 Orderer 节点主要的启动过程。

首先，利用 localmsp，创建签名者结构。

接下来是初始化各个账本结构。如果本地不存在旧的账本文件时，需要初始化系统通道，判断是否需要从外部 genesis 区块文件读入数据，还是根据给定配置初始化 genesis 区块结构。

接下来，新建 grpc server，其中包括 Broadcast() 和 Deliver() 两个服务接口。

最后，启动 grpc 服务。

```
func Start(cmd string, conf *config.TopLevel) {
    signer := localmsp.NewSigner()
    manager := initializeMultichannelRegistrar(conf, signer)
    server := NewServer(manager, signer, &conf.Debug)

    switch cmd {
    case start.FullCommand(): // "start" command
        logger.Infof("Starting %s", metadata.GetVersionInfo())
        initializeProfilingService(conf)
        grpcServer := initializeGrpcServer(conf)
        ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server)
        logger.Info("Beginning to serve requests")
        grpcServer.Start()
    case benchmark.FullCommand(): // "benchmark" command
        logger.Info("Starting orderer in benchmark mode")
        benchmarkServer := performance.GetBenchmarkServer()
        benchmarkServer.RegisterService(server)
        benchmarkServer.Start()
    }
}
```

### initializeGrpcServer

创建 grpc 的服务器。

```
grpcServer, err := comm.NewGRPCServerFromListener(lis, secureConfig)
if err != nil {
logger.Fatal("Failed to return new GRPC server:", err)
}
```

## initializeMultichannelRegistrar

这一部分是十分核心的初始化步骤。

初始化一个 multichannel.Registrar 结构，是整个服务的访问和控制核心结构。

```
type Registrar struct {
    chains          map[string]*ChainSupport
    consenters      map[string]consensus.Consenter
    ledgerFactory   ledger.Factory
    signer          crypto.LocalSigner
    systemChannelID string
    systemChannel   *ChainSupport
    templator       msgprocessor.ChannelConfigTemplator
}
```

通过 createLedgerFactory() 初始化账本结构，包括 file、json、ram 等类型。file 的话会在本地指定目录（/var/production/chains）下创建账本结构。账本所关联的链名称会自动命名为 chain_FILENAME。之后通过指定初始区块，或自动生成来初始化区块链结构。至此，账本结构初始化完成。

接下来，初始化 consenter 部分，初始化 solo、kafka 两种类型。

账本、consenter，再加上传入的签名者结构，通过 multichannel.NewRegistrar() 方法构造一个 multichannel.Registrar 结构，负责处理消息。

multichannel.NewRegistrar() 方法会查看本地已有的链结构文件（例如重启后），并挨个执行如下过程：

- 创建账本工厂对象；
- 获取配置区块，根据配置区块创建账本资源对象；
- 如果配置中包括联盟信息，说明该链结构是系统链，调用。
- 如果配置中不包括联盟信息，说明该链结构是应用链，同样调用 newChainSupport() 方法生成链支持结构，并通过链的 start() 方法启动。

## NewServer

新建一个 Server 结构，包括一个 broadcast 的处理句柄，以及一个 deliver 的处理句柄。

```
type server struct {
    bh broadcast.Handler
    dh deliver.Handler
}
```

NewServer 分别初始化这两个句柄，挂载上前面初始化的 multichannel.Registrar 结构。broadcast 句柄还需要初始化一个配置更新的处理器，负责处理 CONFIG_UPDATE 交易。

```
func NewServer(ml multichain.Manager, signer crypto.LocalSigner) ab.AtomicBroadcastSer
ver {
    s := &server{
        dh: deliver.NewHandlerImpl(deliverSupport{Manager: ml}),
        bh: broadcast.NewHandlerImpl(broadcastSupport{
            Manager:               ml,
            ConfigUpdateProcessor: configupdate.New(ml.SystemChannelID(), configUpdate
Support{Manager: ml}, signer),
        }),
    }
    return s
}
```

## server.go

## util.go

# util 包

## net.go

# consensus 包

# kafka 包

跟 Kafka 集群打交道的接口。

## chain.go

## channel.go

## config.go

## consenter.go

## logger.go

## partitioner.go

## retry.go

## solo 包

## consensus.go

# consensus.go

# mocks

## common 包

## blockcutter 包

## multichannel 包

# util

## util.go

# sample_clients

# broadcast_config

## client.go

## newchain.go

# broadcast_msg 包

## client.go

# deliver_stdout

## client.go

# sbft 包

## backend 包

# simplebft 包

# main.go

主入口文件。

调用 `orderer.common.server` 包中的 `Main()` 方法。

```
func main() {
    server.Main()
}
```

# peer

主命令模块。

作为服务端时候，支持 node 子命令；作为命令行时候，支持 chaincode、channel 等子命令。

作为命令行时候，会维持一个 ChaincodeCmdFactory 结构。

```
type ChaincodeCmdFactory struct {
    EndorserClient  pb.EndorserClient
    Signer          msp.SigningIdentity
    BroadcastClient common.BroadcastClient
}
```

其中：

- EndorserClient 是跟 `peer.address` 指定地址通信的 grpc 通道；
- Signer 为 LocalMSP 中的默认签名实体；
- BroadcastClient 是连接到通过 `-o` 指定的 orderer 服务的 grpc 通道。

# chaincode

包括 install、instantiate、invoke、query 等命令，大家的过程都是类似的，创建 Proposal，发给 peer，获取 Response。

instantiate、upgrade、invoke 等子命令还需要根据 Response 创建 SignedTX，发送给 Orderer。

package、signpackage 子命令则是本地操作。

各子命令的全局参数支持情况如下。

| 命令 | -C 通道 | -c cc 参数 | -E escc | -n 名称 | -o Orderer | -p 路径 | -P policy | -v 版本 | -V vscc |
|---|---|---|---|---|---|---|---|---|---|
| install | 不支持 | 支持 | 不支持 | 必需 | 不支持 | 必需 | 不支持 | 必需 | 不支持 |
| instantiate | 必需 | 必需 | 支持 | 必需 | 支持 | 不支持 | 支持 | 必需 | 支持 |
| upgrade | 必需 | 必需 | 支持 | 必需 | 支持 | 不支持 | 不支持 | 必需 | 支持 |
| package | 不支持 | 支持 | 不支持 | 必需 | 不支持 | 必需 | 不支持 | 必需 | 不支持 |
| invoke | 支持 | 必需 | 不支持 | 必需 | 支持 | 不支持 | 不支持 | 不支持 | 不支持 |
| query | 支持 | 必需 | 不支持 | 必需 | 不支持 | 不支持 | 不支持 | 不支持 | 不支持 |

- 不支持：代表该参数即便被指定，也不会被使用。
- 支持：代表该参数可以被使用，如果不指定，则可能采取默认值或自动获取。
- 必需：该参数必需被指定。

# chaincode.go

# common.go

提供一些通用方法。

## chaincodeInvokeOrQuery

- 获取链码详细信息（含有发出的指令里的信息，比如-C 通道、-n 名字、-c 参数）。
- ChaincodeInvokeOrQuery()（创建proposal，对proposal进行签名，给endorser去执行proposal，拿到proposalResponse后组合成完整的签名交易，类型是一个Envelope，将envelope发给orderer去排序，返回proposalResponse）。
- 检查proposalResponse。
- 返回nil。

# install.go

响应 `peer chaincode install` 命令，将智能合约源码和环境传输到指定的 peer 节点上，并生成智能合约的部署打包文件（name.version）到默认的 `/var/hyperledger/production/chaincodes/` 目录下。

例如

```
peer chaincode install -n test_cc -p github.com/hyperledger/fabric/examples/chaincode/
go/chaincode_example02 -v 1.0
```

命令调用 chaincodeInstall 方法。

一种是通用的方式，通过传入的参数进行打包；一种是直接读入传入的打包文件 ccpackfile 进行处理。

整体流程如下：

- 首先会调用 InitCmdFactory，初始化 Endosermentclient、Signer 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。
- 调用 chaincodeInstall 方法，解析命令行参数，生成 ChaincodeSpec；
- 根据 CS，结合 chaincode 相关数据生成一个 ChaincodeDeploymentSpec（CDS）结构（chainID 为空），并传入 install 方法；
- install 方法基于传入的 CDS，生成一个 install 类型的 Proposal，进行签名和转化为一个 protobuf 提案消息；
- 通过 EndorserClient 经由 grpc 通道发送给 peer 进行背书。

整体流程如下图所示。

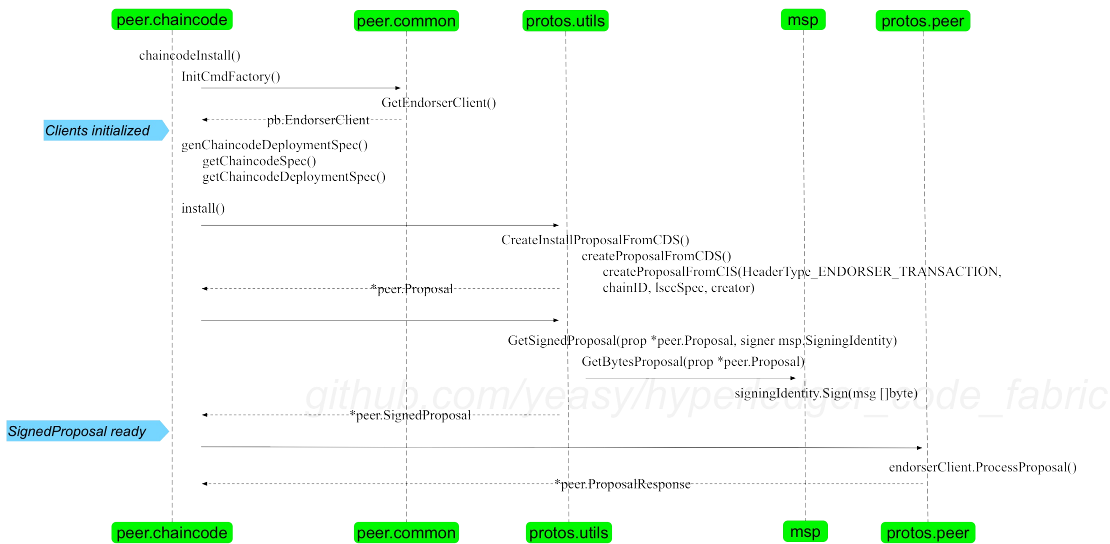





图 *1.20.1.3.1 - peer chaincode install* 过程

## 生成 ChaincodeDeploymentSpec 结构

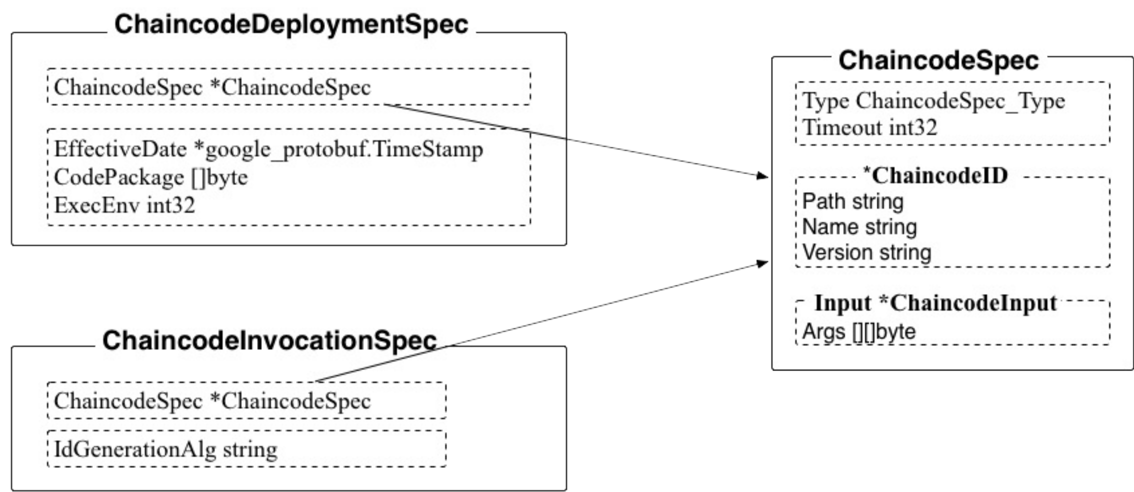



图 *1.20.1.3.2 - ChaincodeDeploymentSpec* 结构

其中，ChaincodeDeploymentSpec 结构的 CodePackage 变量包括所调用的 chaincode 的代码和所需要的环境代码（例如整个 $GOPATH/src 目录下数据），为 tar 格式的二进制数据。

## 进行 endorsement

调用 install 方法，根据 CDS 来生成 Install Proposal。

首先，从本地 MSP 中拿到签名体身份（签名体在初始化阶段完成导入，包括证书和私钥）。

首先，创建 Install Proposal。

```
prop, _, err := utils.CreateInstallProposalFromCDS(msg, creator)
```

实际调用的是 protos/utils/proputils.go 中的 createProposalFromCDS 方法，创建一个对 LSCC 的 ChaincodeInvocationSpec，然后基于它创建一个 Proposal 结构。在此期间，需要生成 transaction id（随机生成的 nonce 值和 creator 信息，一起进行摘要）。

Proposal 结构体内容被 cf.Signer 进行签名，生成 SignedProposal 消息。SignedProposal 通过 endorsement 客户端通过 grpc 发送到 peer 节点进行背书，正常会收到 ProposalResponse 消息。

# instantiate.go

响应 `peer chaincode instantiate` 命令，生成智能合约容器，在 peer 节点上启动。

例如

```
peer chaincode instantiate -n test_cc -c '{"Args":["init","a","100","b","200"]}' -o orderer0:7050 -v 1.0
```

instantiate 支持包括 policy、channel、escc、vscc 在内更多的命令行参数。

命令调用 chaincodeDeploy 方法。

首先会调用 InitCmdFactory 方法，初始化 EndosermentClient、Signer、BroadcastClient 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。

之后，会调用 instantiate 方法，instantiate 方法的流程如下：

- 根据传入的各种参数，生成 ChaincodeSpec，注意 instantiate 和 upgrade 是支持 policy、escc、vscc 等参数。
- 生成 ChaincodeDeploymentSpec 结构。
- 根据 CDS、签名实体、策略、通道、escc、vscc 等信息，创建一个 LSCC 的 ChaincodeInvocationSpec，根据这个 CIS，添加上 TxID（随机数+签名实体，进行 hash），创建一个 Proposal 出来。
- 根据签名实体，对 Proposal 进行签名。
- 调用 EndorserClient，发送 gprc 消息，将签名后 Proposal 发给指定的 peer。
- 根据 peer 的返回，创建一个 Envelop 结构并进行签名。
- 将 Envelop 通过 grpc 通道发给 orderer。

整体流程如下图所示。

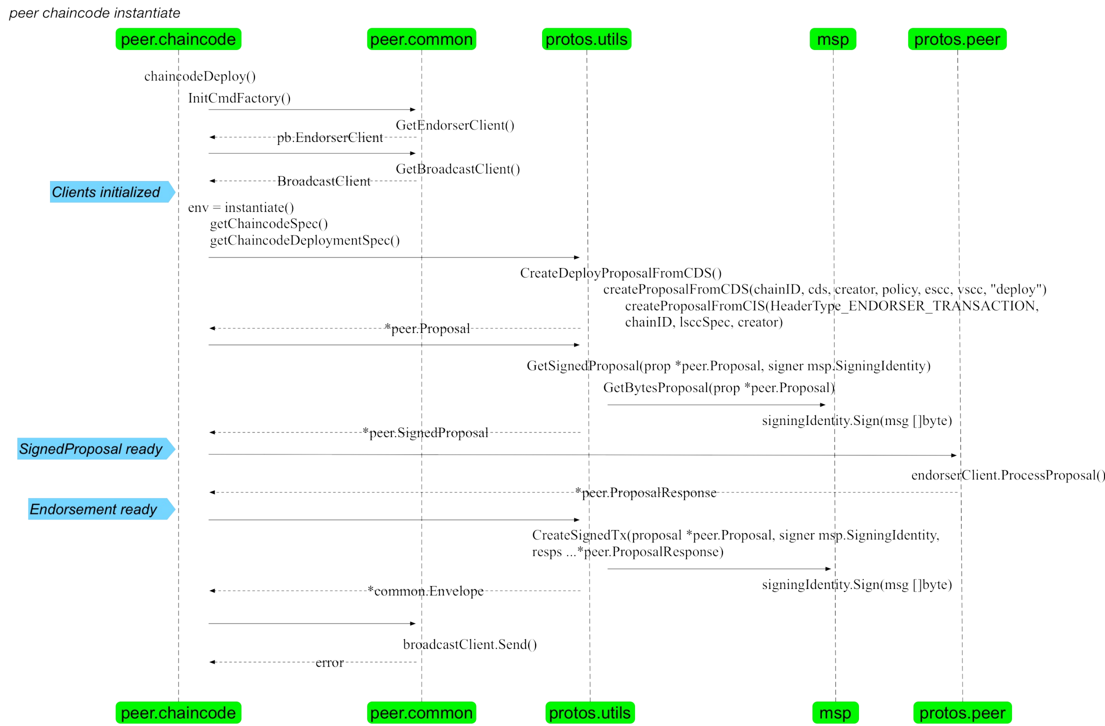



图 *1.20.1.4.1 - peer chaincode instantiate* 过程

# invoke.go

响应 `peer chaincode invoke` 命令，执行某个 chaincode 中操作。

例如

```
peer chaincode invoke -n test_cc -c '{"Args":["invoke","a","b","10"]}' -o orderer0:7050
```

命令会调用 chaincodeQuery。

首先会调用 InitCmdFactory，初始化 Endosermentclient、Signer 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 chaincodeInvokeOrQuery 方法。

chaincodeInvokeOrQuery 方法主要过程如下：

- 生成 ChaincodeSpec。
- 根据 CS、chainID、签名实体等，生成 ChaincodeInvocationSpec。
- 根据 CIS，生成 Proposal，并进行签名。
- 签名后，通过 endorserClient 发送给指定的 peer。
- 利用获取到的 Response，创建 SignedTX，发送给 orderer。
- 成功的话，输出成功消息，注意 invoke 是异步操作，无法获取到执行结果。

注意 invoke 和 query 的区别，query 不需要创建 SignedTx 发送到 orderer，而且会返回结果。

# java.go

# list.go

# nojava.go

# package.go

响应 `peer chaincode package` 命令，将某个 chaincode 打包为 deployment 的 spec。

例如

```
$ peer chaincode package -n test_cc -c '{"Args":["init","a","100","b","200"]}' -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v 1.0  test_cc_1.0.pkg
```

命令会调用 chaincodePackage 方法。

首先会调用 InitCmdFactory，初始化 Signer 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。

之后主要过程如下：

- 根据传入的各种参数，生成 ChaincodeSpec。
- 生成 ChaincodeDeploymentSpec 结构。
- 根据 CDS 等创建 签名后的 ChaincodeDeploymentSpec，并封装为 Envelop 结构（其中数据是一个 SignedChaincodeDeploymentSpec）。
- 将 Envelop 结构写到本地指定的文件中。

# query.go

响应 `peer chaincode query` 命令，查询某个 chaincode 中变量。

例如

```
peer chaincode query -n test_cc -c '{"Args":["query","a"]}'
```

命令会调用 chaincodeQuery。

首先会调用 InitCmdFactory，初始化 Endosermentclient、Signer 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 chaincodeInvokeOrQuery 方法。

chaincodeInvokeOrQuery 方法主要过程如下：

- 生成 ChaincodeSpec。
- 根据 CS、chainID、签名实体等，生成 ChaincodeInvocationSpec。
- 根据 CIS，生成 Proposal，并进行签名。
- 签名后，通过 endorserClient 发送给指定的 peer。
- 成功的话，获取到 ProposalResponse，打印出 proposalResp.Response.Payload。

注意 invoke 和 query 的区别，query 不需要创建 SignedTx 发送到 orderer，而且会返回结果。

# signpackage.go

# upgrade.go

响应 `peer chaincode upgrade` 命令，升级某个 chaincode 到新的版本。

例如

```
$ peer chaincode upgrade -n test_cc  -o orderer0:7050 -c '{"Args":["init","a","100","b
","200"]}' -v 1.1
```

命令会调用 chaincodeUpgrade。

首先会调用 InitCmdFactory，初始化 Endosermentclient、Signer、OrdererClient 等结构。这一步对于所有 chaincode 子命令来说都是类似的，个别会初始化不同的结构。

之后调用 upgrade 方法。

upgrade 方法主要过程如下：

- 根据传入的各种参数，生成 ChaincodeSpec，注意 instantiate 和 upgrade 是支持 policy、escc、vscc 等参数。
- 生成 ChaincodeDeploymentSpec 结构。
- 根据 CDS、签名实体、策略、通道、escc、vscc 等信息，创建一个 LSCC 的 ChaincodeInvocationSpec，根据这个 CIS，添加上 TxID（随机数+签名实体，进行 hash），创建一个 Proposal 出来；
- 根据签名实体，对 Proposal 进行签名。
- 调用 EndorserClient，发送 gprc 消息，将签名后 Proposal 发给指定的 peer。
- 根据 peer 的返回，创建一个 Envelop 结构（SignedTx）并进行签名。
- 将 Envelop 通过 grpc 通道发给 orderer。

# channel

包括 create、fetch、join、list 等命令。

所有命令都会先执行初始化方法 InitCmdFactory，初始化需要的 EndorserClient 或 OrdererClient。

- create：创建一个新的 channel。根据指定的配置交易文件路径或默认配置，创建 Envelope 结构，发送给 Orderer，并将所指定创建通道中的初始区块写到本地文件 chainID.block。
- fetch：获取指定通道的初始区块。从 Orderer 获取指定通道的初始区块，写到本地文件 chainID.block。
- join：让 peer 加入某个通道。读取本地的 block 文件，生成一个 cscc 的 JoinChain 交易 spec，进一步封装为一个 ChaincodeInvocationSpec，创建 CONFIG 类型的 proposal，并签名，通过 EndorserClient 发给 peer。
- list：列出 peer 所加入的所有通道。生成一个 cscc 的 GetChannels 交易 spec，进一步封装为一个 ChaincodeInvocationSpec，创建 ENDORSER_TRANSACTION 类型的 proposal，并签名，通过 EndorserClient 发给 peer。

各子命令的参数支持情况如下。

| 命令 | **-b** 区块文件路径 | **-c chainID** | **-f** 配置交易文件路径 | **-o Orderer** | **--tls** | **--cafile tls** 证书路径 |
|---|---|---|---|---|---|---|
| create | 不支持 | 必需 | 可选 | 必需 | 可选 | 可选 |
| fetch | 不支持 | 必需 | 不支持 | 必需 | 可选 | 可选 |
| join | 必需 | 不支持 | 不支持 | 不支持 | 不支持 | 不支持 |
| list | 不支持 | 不支持 | 不支持 | 不支持 | 不支持 | 不支持 |

# channel.go

`peer channel` 相关命令的入口。

进一步支持 join、create、fetch、list 等子命令。

```
// Cmd returns the cobra command for Node
func Cmd(cf *ChannelCmdFactory) *cobra.Command {

AddFlags(channelCmd)
channelCmd.AddCommand(joinCmd(cf))
channelCmd.AddCommand(createCmd(cf))
channelCmd.AddCommand(fetchCmd(cf))
channelCmd.AddCommand(listCmd(cf))

return channelCmd
}
```

所有 channel 子命令都会先调用 InitCmdFactory 来进行必要的初始化，根据命令需求来生成 endorserClient、BroadcastClient 和 DeliverClient。

# create.go

`peer channel create` 命令的入口。

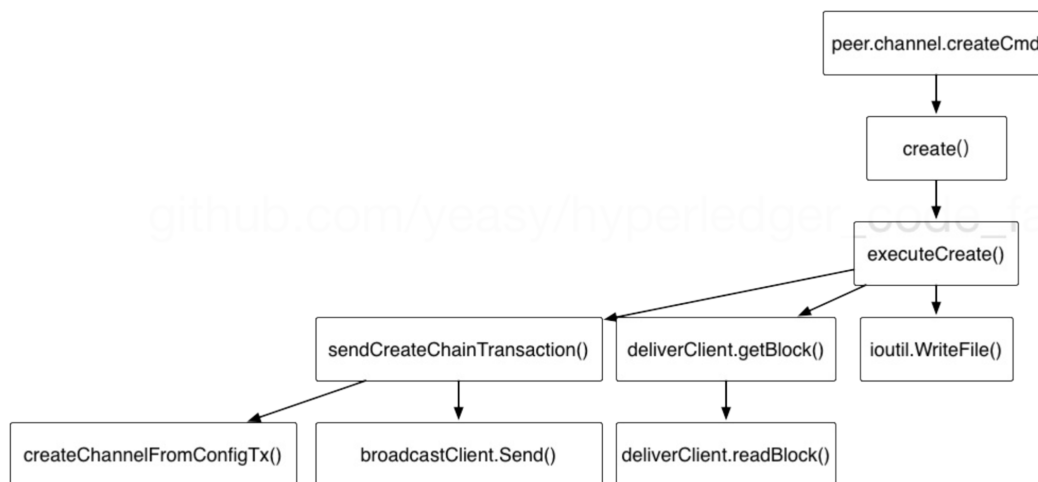



图 *1.20.2.2.1 - peer channel create*

调用 create 方法，首先通过 InitCmdFactory 进行初始化，然后调用 executeCreate。

executeCreate 流程包括：

- 客户端调用 sendCreateChainTransaction，检查指定的配置交易文件，或者利用默认配置，构造一个创建应用通道的配置交易结构，封装为 Envelope，指定类型为 CONFIG_UPDATE。
- 客户端发送配置交易到 Orderer 服务。
- Orderer 收到 CONFIG_UPDATE 消息后，检查指定的通道还不存在，则开始新建过程，构造该应用通道的初始区块。
    - Orderer 首先检查通道应用（Application）配置中的组织都在创建的联盟（Consortium）配置组织中。
    - 之后从系统通道中获取 Orderer 相关的配置，并创建应用通道配置，对应 mod_policy 为系统通道配置中的联盟指定信息。
    - 接下来根据 CONFIG_UPDATE 消息的内容更新获取到的配置信息。所有配置发生变更后版本号都要更新。
    - 最后，发送到系统通道中，完成应用通道的创建过程。
- 客户端从 Orderer 获取到该应用通道的初始区块。
- 客户家将收到的区块写入到本地的 chainID + ".block" 文件。这个文件后续会被需要加入到通道的节点使用。

sendCreateChainTransaction 方法会检查，如果提供了 tx 文件了，则直接读取为 Envelope 结构；如果不存在，则通过默认值来创建一个 Envelope 结构。之后将 Envelope 结构发给 orderer。

Envelope 结构中 Payload.Data 是一个 ConfigUpdateEnvelope 结构。

# deliverclient.go

# fetchconfig.go

# getinfo.go

# join.go

join 过程也十分简单，主要是 peer 完成。

同样的，首先通过 InitCmdFactory 进行初始化。

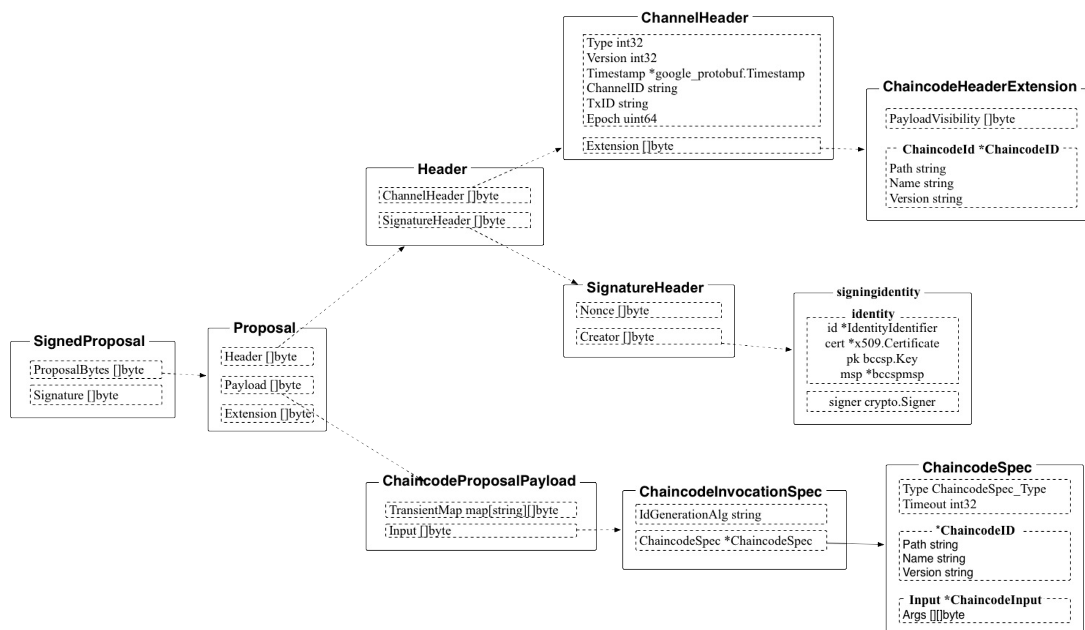



图 *1.20.2.6.1 - Signed Proposal* 结构

- 客户端首先创建一个 ChaincodeSpec 结构，其 input 中的 Args 第一个参数是 CSCC.JoinChain（指定调用配置链码的操作），第二个参数为所加入通道的初始区块。
- 利用 CS 构造一个 ChaincodeInvocationSpec 结构。
- 利用 CIS，创建 Proposal 结构并进行签名，channel 头部类型为 CONFIG。
- 客户端通过 gRPC 将 Proposal 发给 Endorser，调用 ProcessProposal() 方法进行处理，主要通过配置系统链码进行本地链的初始化工作，并通过获取通道的配置来得到 Ordering 服务地址。
- 初始化完成后，即可收到来自通道内的 Gossip 消息等。

# list.go

同样的，首先通过 InitCmdFactory 进行初始化。

主要实现过程如下：

- 客户端首先创建一个 ChaincodeSpec 结构，其 input 中的 Args 第一个参数是 CSCC.GetChannels（指定调用配置链码的操作）。
- 利用 CS 构造一个 ChaincodeInvocationSpec 结构。
- 利用 CIS，创建 Proposal 结构并进行签名，channel 头部类型为 ENDORSER_TRANSACTION。
- 客户端通过 gRPC 将 Proposal 发给 Endorser（所操作的 Peer），调用 ProcessProposal() 方法进行处理，主要是通过配置系统链码查询本地链信息并返回。
- 命令执行成功后，客户端会受到来自 Peer 端的回复消息，从其中提取出应用通道列表信息并输出。

# signconfigtx.go

# update.go

# channel-artifacts 包

# clilogging

# common.go

# getlevel.go

# logging.go

# revertlevels.go

## setlevel.go

# common

# common.go

# mockclient.go

# ordererclient.go

# crypto 包

# gossip

# mocks

## mocks.go

# mcs.go

## sa.go

# node

负责 `peer node` 子命令。

调用其他功能模块来具体实现。

- start：启动节点。
- stop：停止节点。
- status：查看节点状态。

## 启动服务

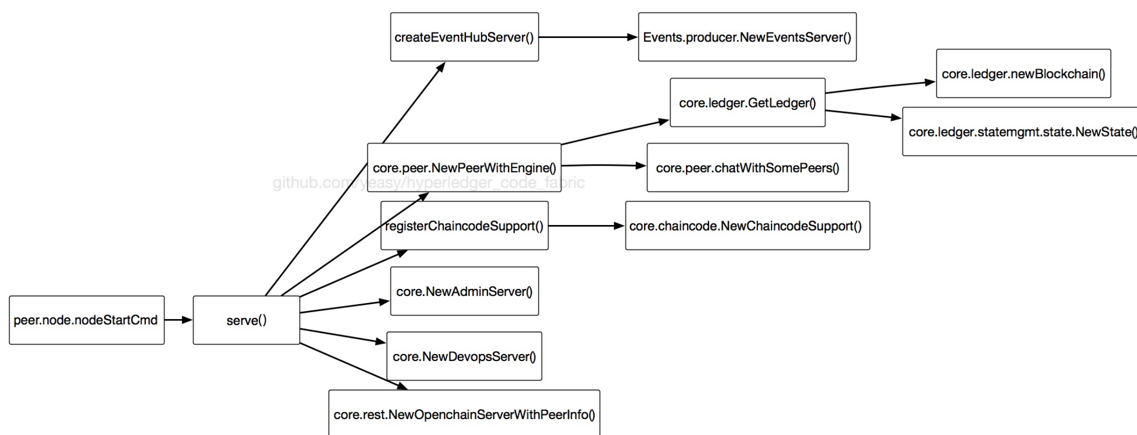





图 *1.20.8.1 - peer node start*

## 查看状态

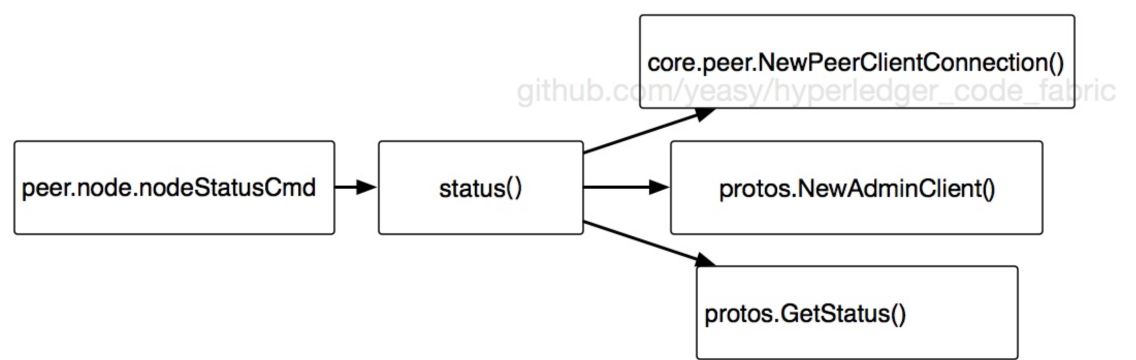



图 *1.20.8.2 - peer node status*

# 停止服务

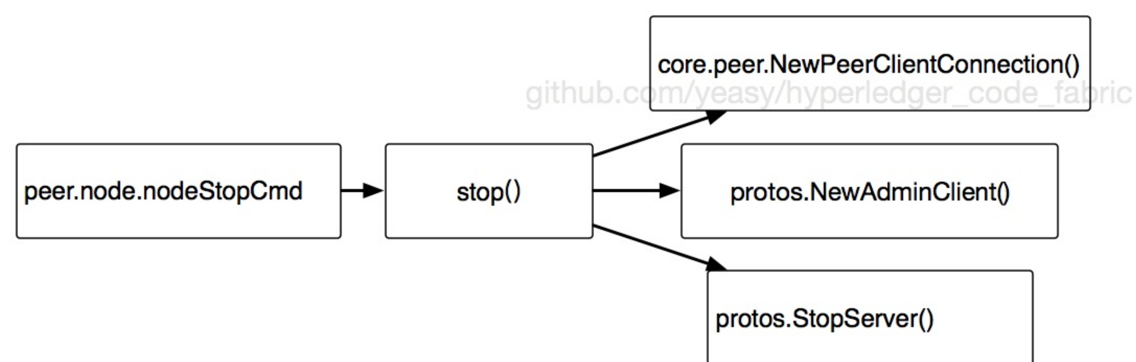



图 *1.20.8.3 - peer node stop*

# node.go

node 子命令进一步支持 start、stop、join、status 等子命令。

```go
// Cmd returns the cobra command for Node
func Cmd() *cobra.Command {
    nodeCmd.AddCommand(startCmd())
    nodeCmd.AddCommand(statusCmd())
    nodeCmd.AddCommand(stopCmd())
    nodeCmd.AddCommand(joinCmd())

    return nodeCmd
}
```

# start.go

负责 `peer node start` 命令。

最重要的是 `func serve(args []string) error` 函数，启动一个节点服务，主要是启动各个 GRPC 的服务端。包括 EventsServer 服务、chaincodesupport 服务、admin 服务、endorser 服务、gossip 服务。

- EventsServer 服务（events.producer.EventsServer）：提供 Chat GRPC 调用。
- ChaincodeSupport（core.chaincode.ChaincodeSupport）服务：提供 Execute、Launch、Register、Stop 等方法。
- ServerAdmin 服务（core.ServerAdmin）：提供 GetStatus、StartServer、StopServer、GetModuleLogLevel、SetModuleLogLevel 等方法。
- Endorser 服务（core.endorser.Endorser）：提供 ProcessProposal 方法。
- GossipService 服务（gossip.service.GossipService）：提供 NewConfigEventer、InitializeChannel、GetBlock、AddPayload 方法。

startCmd() 方法调用 serve() 方法。

整体流程如下图所示。

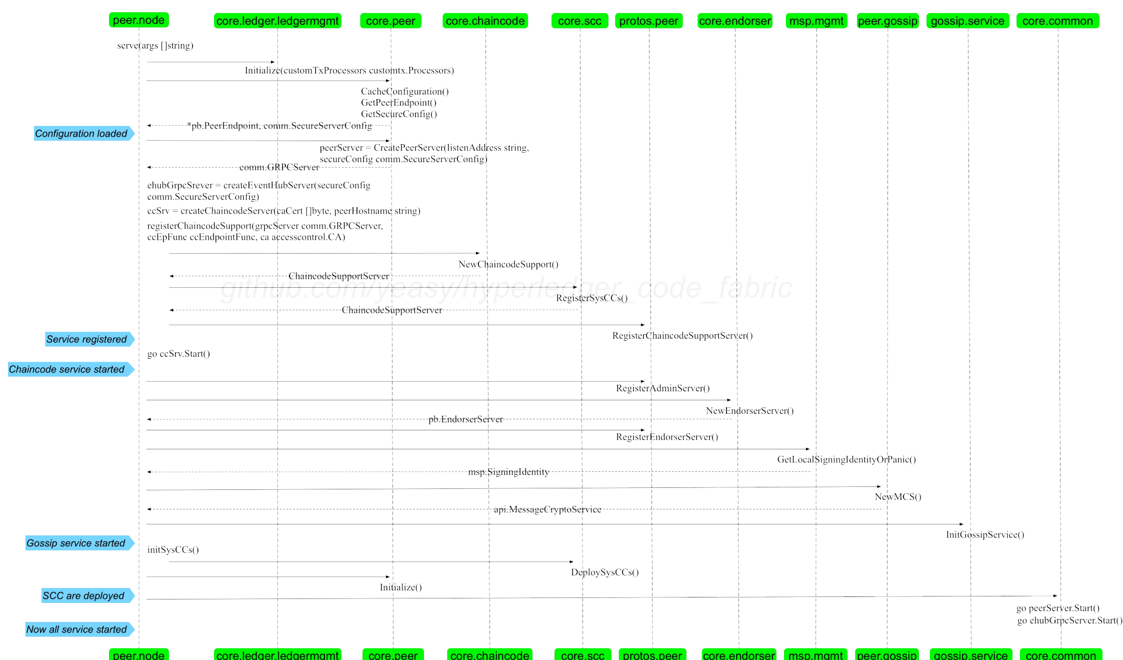




图 *1.20.8.2.1 - peer node start* 整体流程

## 配置读取和缓存

首先是进行配置管理，根据配置信息和一些计算来构建 cache 结构，探测节点信息等。主要调用 core.peer 包来实现。

```
    if err := peer.CacheConfiguration(); err != nil {
        return err
    }

    peerEndpoint, err := peer.GetPeerEndpoint()
    if err != nil {
        err = fmt.Errorf("Failed to get Peer Endpoint: %s", err)
        return err
    }
```

## 创建 eventHub 服务

eventHub 服务监听到 7053 端口，仅在 VP 节点上打开。

调用 `createEventHubServer` 方法实现，主要过程为：

创建 EventHub 服务，通过调用 createEventHubServer() 方法来实现，该服务也是 grpc，只有 vp 节点才开启。

```
    lis, err = net.Listen("tcp", viper.GetString("peer.validator.events.address"))
    if err != nil {
        return nil, nil, fmt.Errorf("failed to listen: %v", err)
    }

    //TODO - do we need different SSL material for events ?
    var opts []grpc.ServerOption
    if comm.TLSEnabled() {
        creds, err := credentials.NewServerTLSFromFile(viper.GetString("peer.tls.cert.
file"), viper.GetString("peer.tls.key.file"))
        if err != nil {
            return nil, nil, fmt.Errorf("Failed to generate credentials %v", err)
        }
        opts = []grpc.ServerOption{grpc.Creds(creds)}
    }

    grpcServer = grpc.NewServer(opts...)
    ehServer := producer.NewEventsServer(uint(viper.GetInt("peer.validator.events.buff
ersize")), viper.GetInt("peer.validator.events.timeout"))
    pb.RegisterEventsServer(grpcServer, ehServer)
```

eventHub 服务支持的方法为 Chat。

```
type EventsServer interface {
    // event chatting using Event
    Chat(Events_ChatServer) error
}
```

## 创建和注册 grpc 服务

创建 gprc 服务，并注册上 chaincode、admin、endorser、gossip 等服务，并初始化注册 chainless 系统 chaincode 和创建初始区块。

```
grpcServer := grpc.NewServer(opts...)

registerChaincodeSupport(grpcServer)

logger.Debugf("Running peer")

// Register the Admin server
pb.RegisterAdminServer(grpcServer, core.NewAdminServer())

// Register the Endorser server
serverEndorser := endorser.NewEndorserServer()
pb.RegisterEndorserServer(grpcServer, serverEndorser)

// Initialize gossip component
bootstrap := viper.GetStringSlice("peer.gossip.bootstrap")
service.InitGossipService(peerEndpoint.Address, grpcServer, bootstrap...)
defer service.GetGossipService().Stop()
```

其中，chaincode 服务支持方法为

```
type ChaincodeSupportServer interface {
    Register(ChaincodeSupport_RegisterServer) error
}
```

admin 服务支持方法为

```
type AdminServer interface {
    // Return the serve status.
    GetStatus(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    StartServer(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    StopServer(context.Context, *google_protobuf1.Empty) (*ServerStatus, error)
    GetModuleLogLevel(context.Context, *LogLevelRequest) (*LogLevelResponse, error)
    SetModuleLogLevel(context.Context, *LogLevelRequest) (*LogLevelResponse, error)
}
```

endorser 服务支持方法为

```
type EndorserServer interface {
    ProcessProposal(context.Context, *SignedProposal) (*ProposalResponse, error)
}
```

gossip 服务支持方法为

```
type GossipServer interface {
    // GossipStream is the gRPC stream used for sending and receiving messages
    GossipStream(Gossip_GossipStreamServer) error
    // Ping is used to probe a remote peer's aliveness
    Ping(context.Context, *Empty) (*Empty, error)
}
```

## 启动 grpc 服务和 eventHub 服务

之后是启动 grpc 服务，监听到 7051 端口。

```
go func() {
        var grpcErr error
        if grpcErr = grpcServer.Serve(lis); grpcErr != nil {
            grpcErr = fmt.Errorf("grpc server exited with error: %s", grpcErr)
        } else {
            logger.Info("grpc server exited")
        }
        serve <- grpcErr
    }()
```

启动 eventHub 服务。

```
if ehubGrpcServer != nil && ehubLis != nil {
        go ehubGrpcServer.Serve(ehubLis)
    }
```

最后，如果需要 profiling，还会打开监听服务。

# status.go

负责 `peer node status` 命令。

主要包括 status 方法，通过 admin 服务获取 peer 状态。

```
func status() (err error) {

    adminClient, err := common.GetAdminClient()
    if err != nil {
        logger.Warningf("%s", err)
        fmt.Println(&pb.ServerStatus{Status: pb.ServerStatus_UNKNOWN})
        return err
    }

    status, err := adminClient.GetStatus(context.Background(), &empty.Empty{})
    if err != nil {
        logger.Infof("Error trying to get status from local peer: %s", err)
        err = fmt.Errorf("Error trying to connect to local peer: %s", err)
        fmt.Println(&pb.ServerStatus{Status: pb.ServerStatus_UNKNOWN})
        return err
    }
    fmt.Println(status)
    return nil
}
```

# scripts 包

# version

# version.go

# log.txt

# main.go

主服务，所有 peer 命令都从这里入口。

各个子命令的处理在对应子包中，如

- node：node 对应子命令
- chaincode：chaincode 对应子命令
- channel：channel 对应子命令

main 函数主要完成子命令的注册和一些初始化配置工作，为执行子命令准备好环境，包括：

- 调用 InitConfig() 从本地的 yaml、环境变量以及命令行选项中读取 Peer 命令相关的配置信息；
- 之后注册各个子命令；
- 最后调用 InitCrypto() 从本地读入 MSP 配置文件和 BCCSP 配置，初始化 MSP 部分。初始化会创建本地的一个 bccspmsp 结构（msp.mspimpl.go），然后利用本地读取的 MSP（包括各种证书和私钥等） 和 BCCSP 配置进行初始化。

peer 的 MSP 文件路径从 `peer.mspConfigPath` 变量读取（相对路径，前面会拼接上配置文件路径，默认为 $FABRIC_CFG_PATH/msp）；默认的 mspID 是 `peer.localMspId`（DEFAULT）。BCCSP 配置从 `peer.BCCSP` 中读取。

## MSP 初始化

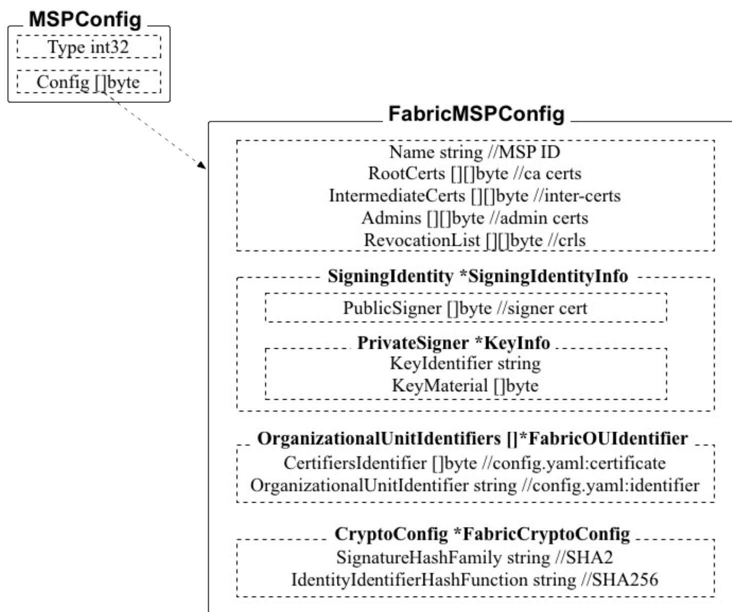



图 *1.20.12.1 - MSP* 配置结构

通过 peer/common/common.go 文件中的 InitCrypto 方法完成。

```
func InitCrypto(mspMgrConfigDir string, localMSPID string) error {
    // Init the BCCSP
    var bccspConfig *factory.FactoryOpts
    err := viperutil.EnhancedExactUnmarshalKey("peer.BCCSP", &bccspConfig)
    if err != nil {
        return fmt.Errorf("Could not parse YAML config [%s]", err)
    }

    err = mspmgmt.LoadLocalMsp(mspMgrConfigDir, bccspConfig, localMSPID)
    if err != nil {
        return fmt.Errorf("Fatal error when setting up MSP from directory %s: err %s\n"
, mspMgrConfigDir, err)
    }

    return nil
}
```

其中，默认的 BCCSP 配置从 `peer.BCCSP` 中读取，示例配置为

```
BCCSP:
        Default: SW
        SW:
            # TODO: The default Hash and Security level needs refactoring to be
            # fully configurable. Changing these defaults requires coordination
            # SHA2 is hardcoded in several places, not only BCCSP
            Hash: SHA2
            Security: 256
            # Location of Key Store, can be subdirectory of SbftLocal.DataDir
            FileKeyStore:
                # If "", defaults to 'mspConfigPath'/keystore
                # TODO: Ensure this is read with fabric/core/config.GetPath() once rea
dy
                KeyStore:
```

通过 msp/mgmt/mgmt.go 中的 LoadLocalMsp 方法来导入 MSP 相关的文件和配置。LoadLocalMsp 会先读入各种文件和初始化配置，然后按照配置进行初始化。

首先，调用 msp/cofnigbuilder.go 中的 GetLocalMspConfig 方法，进一步调用 getMspConfig 方法。getMspConfig 方法导入所有提供的 PEM 格式的证书和密钥文件，并从 MSP 配置目录下查找 config.yaml 并读取 OUIdentifier 信息。

之后通过 `GetLocalMSP().Setup(conf)` 进行配置。GetLocalMSP()会通过 msp.NewBccspMsp()生成一个 msp.bccspmsp 对象。之后，调用 msp/mspimpl.go 中的 Setup 方法，将读入的证书文件等配置写到 msp.bccspmsp 对象中。

至此，完成 MSP 的初始化工作。

# nohup.out

# proposals

# r1

# protos

Protobuf 格式的数据结构和消息协议。都在同一个 protos 包内。

这里面是所有基本的数据结构（message）定义和 GRPC 的服务（service）接口声明。

所有的 `.proto` 文件是 protobuf 格式的声明文件，`.pb.go` 文件是基于 `.proto` 文件生成的 go 语言的类文件。

protobuf 工具可以从 这里 下载，推荐使用 3.0 版本系列。

下载安装后，需要安装对应语言的编译器，例如要生成 go 语言代码，则需要安装 `protoc-gen-go`。

```
$ go get github.com/golang/protobuf/protoc-gen-go
```

可以使用 protobuf 编译器基于 protobuf 模板文件来生成各种语言的类文件。

```
$ protoc \
--proto_path=IMPORT_PATH \
--cpp_out=DST_DIR \
--java_out=DST_DIR \
--python_out=DST_DIR \
--go_out=DST_DIR \
--ruby_out=DST_DIR \
--javanano_out=DST_DIR \
--objc_out=DST_DIR \
--csharp_out=DST_DIR \
path/to/file.proto
```

其中，`--proto_path=IMPORT_PATH` 是当 proto 文件中存在导入时候，进行查找的路径，等价于 `-I=IMPORT_PATH`，可以多次使用来指定多个导入路径。

为了生成支持 grpc 的代码，还可以提供生成参数 `plugins=grpc`，例如

```
$ protoc \
--proto_path=IMPORT_PATH \
--go_out=plugins=grpc:DST_DIR \
path/to/file.proto
```

另外，生成的结构体，一般都至少默认支持 4 个默认生成的方法。

- Reset()：重置结构体。
- String() string：返回代表对象的字符串。

- ProtoMessage()：协议消息。
- Descriptor([]byte, []int)：描述信息。

# common

# block.go

# collection.pb.go

# collection.proto

# common.go

# common.pb.go

# common.proto

# configtx.go

# configtx.pb.go

# configtx.proto

# configuration.go

# configuration.pb.go

# configuration.proto

# ledger.pb.go

# ledger.proto

# policies.go

# policies.pb.go

# policies.proto

# signed_data.go

# gossip

## extensions.go

# message.pb.go

# message.proto

# idemix 包

# idemix.proto

# ledger

# queryresult

# kv_query_result.pb.go

## kv_query_result.proto

# rwset

## kvrwset

## tests 包

# rwset.pb.go

# rwset.proto

# msp

# identities.pb.go

# identities.proto

# msp_config.go

# msp_config.pb.go

# msp_config.proto

# msp_principal.go

# msp_principal.pb.go

# msp_principal.proto

# orderer

## ab.pb.go

## ab.proto

# configuration.go

# configuration.pb.go

# configuration.proto

# kafka.pb.go

# kafka.proto

# peer

## admin.pb.go

# admin.proto

# chaincode.pb.go

# chaincode.proto

# chaincode_event.pb.go

# chaincode_event.proto

# chaincode_shim.pb.go

# chaincode_shim.proto

# chaincodeunmarshall.go

# configuration.go

# configuration.pb.go

# configuration.proto

# events.pb.go

# events.proto

# init.go

# peer.pb.go

# peer.proto

# proposal.go

# proposal.pb.go

# proposal.proto

# proposal_response.go

# proposal_response.pb.go

# proposal_response.proto

# query.pb.go

# query.proto

## resources.go

# resources.pb.go

# resources.proto

# signed_cc_dep_spec.pb.go

# signed_cc_dep_spec.proto

# transaction.go

# transaction.pb.go

# transaction.proto

# testutils

# txtestutils.go

# utils

包含了基本的数据结构的使用（可以理解为数据的编码和解码）使用的是 `github.com/golang/protobuf/proto` 下的 `Marshal()` 和 `Unmarshal()` 方法

# blockutils.go

# commonutils.go

主要的函数：

字节数据解码为Payload：UnmarshalPayload([]byte) return：*cb.Payload, err

字节数据解码为Envelope：UnmarshalEnvelope([]byte) return：*cb.Envelope, err

从区块中提取指定序号的Envelope：ExtractEnvelope(*cb.Block, int) return*：cb.Envelope, err

从Envelope中提取Payload：ExtractPayload(*cb.Envelope) return*：cb.Payload, err

字节数据解码为ChannelHeader：UnmarshalChannelHeader([]byte) return：*cb.ChannelHeader, err

字节数据解码为ChaincodeID：UnmarshalChaincodeID([]byte) return：*cb.ChaincodeID, err

对消息签名：SignOrPanic(*LocalSigner, []byte) return：[]byte Ps：实际上LocalSigner的Sign()方法直接返回原数据

判断区块是否包含配置更新交易：IsConfigBlock(*cb.Block) return：bool 解释：提取块的第一个Envelope，提取该Envelope的Payload，解码ChannelHeader，判断类型

其他的函数：创建新ChannelHeader、SignatureHeader等

# proputils.go

## CreateChaincodeProposal

- 节点身份证书序列化后与随机数计算出交易ID。
- 构建ChaincodeHeaderExtension（含有ChaincodeID）。
- 构建ChaincodeProposalPayload（含有ChaincodeSpec）。
- 获取事件戳。
- 构建Header。
- 返回Proposal。

# txutils.go

## CreateSignedTx(proposal, signer, resps)

- 按照下图的结构封装交易（其中有很重要的一步是按字节比对所有的 `ProposalResponse.Payload` 是否相等，处理多个Endorser情况下出现模拟结果不一样的问题，直接返回nil）。

Ps. 传入的resps是一个 `ProposalResponse` 的数组。

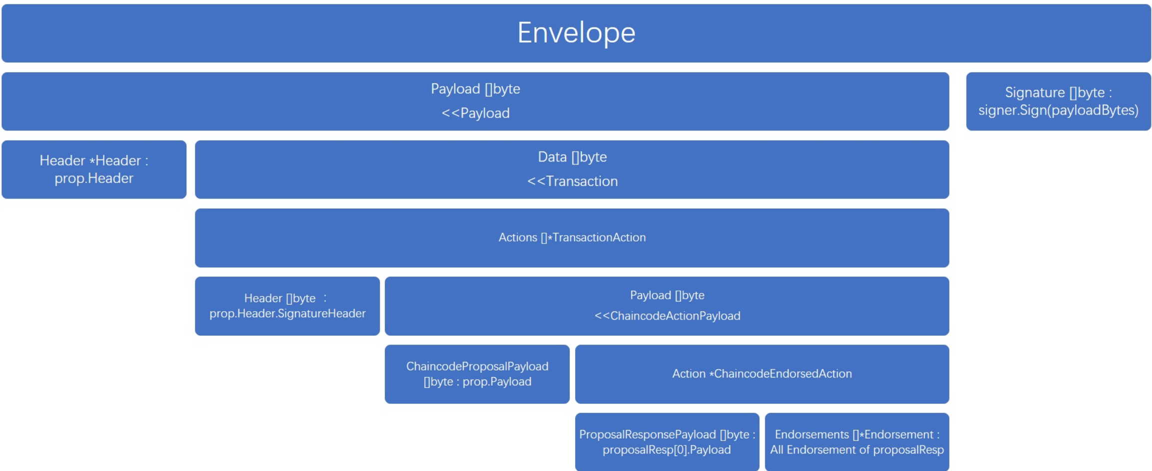



图 *1.22.9.4.1* - 交易结构

# release

# templates

## get-docker-images.in

# release_notes

# v1.0.0-rc1.txt

# v1.0.0.txt

# v1.0.1.txt

# v1.0.2.txt

# v1.0.3.txt

# v1.1.0-preview.txt

# sampleconfig

提供了一些样例证书文件和配置文件。

pem （Privacy Enhanced Mail，属于 X.509 证书标准格式）格式，内容是 BASE64 编码，可以通过 openssl 来查看内容。

如

```
$ openssl x509 -in msp/sampleconfig/admincerts/admincert.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:70:93:0c:e5:38:ee:c5:02:e4:ae:24:9f:f0:9a:aa:78:75:d7:86
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=California, L=San Francisco, O=Internet Widgets, Inc., OU=WWW
, CN=example.com
        Validity
            Not Before: Oct 12 19:31:00 2016 GMT
            Not After : Oct 11 19:31:00 2021 GMT
        Subject: C=US, ST=California, L=San Francisco, O=Internet Widgets, Inc., OU=WW
W, CN=example.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
            EC Public Key:
                pub:
                    04:a2:07:e5:bd:89:69:29:aa:89:05:c7:ca:a0:be:
                    72:69:27:20:27:05:8b:90:1d:75:58:ec:42:2c:c3:
                    57:ab:99:e2:fe:ac:f8:f5:a2:27:2c:df:03:fa:d3:
                    b4:cb:d8:00:fa:bf:c9:e1:07:a4:15:01:d6:3d:39:
                    de:6c:67:a4:cb
                ASN1 OID: prime256v1
        X509v3 extensions:
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Subject Key Identifier:
                17:67:42:3D:AA:9E:82:3F:C4:C5:1D:9F:5B:C3:99:D1:B5:9C:48:10
            X509v3 Authority Key Identifier:
                keyid:17:67:42:3D:AA:9E:82:3F:C4:C5:1D:9F:5B:C3:99:D1:B5:9C:48:10

    Signature Algorithm: ecdsa-with-SHA256
        30:44:02:20:07:a7:94:5b:a7:d1:26:d4:6f:d4:98:a9:fc:5a:
        c0:9b:a8:37:d6:79:c5:5b:64:f4:23:dd:12:b8:a0:6e:64:41:
        02:20:40:07:b8:38:e2:98:84:97:61:dd:fe:d4:45:a2:9f:19:
        37:f8:f7:6f:e7:99:19:ad:2b:ec:92:2a:3a:47:4a:b5
```

# msp

# admincerts

# admincert.pem

# cacerts

## cacert.pem

# keystore

## key.pem

# signcerts

## peer.pem

# tlscacerts

## tlsroot.pem

## tlsintermediatecerts 包

## tlsintermediate.pem

# config.yaml

# configtx.yaml

# core.yaml

peer 节点相关的样例配置。

# orderer.yaml

# scripts

一些辅助脚本，多数为外部 Makefile 调用。

# bootstrap-1.0.0-alpha2.sh

# bootstrap-1.0.0-beta.sh

# bootstrap-1.0.0-rc1.sh

# bootstrap-1.0.0.sh

# bootstrap-1.0.1.sh

# bootstrap-1.0.2.sh

# bootstrap-1.0.3.sh

# bootstrap-1.1.0-preview.sh

# changelog.sh

# check_license.sh

# check_spelling.sh

# compile_protos.sh

找到所有的 `.proto` 文件，编译生成支持 grpc 的 `.go` 文件。

# containerlogs.sh

将本地的日志文件上传到 chunk.io，方便进一步分析。

# foldercopy.sh

clone 远端给定用户仓库下的 fabric 代码到本地的 Go 路径下。

# goListFiles.sh

列出所有的包，检查导入路径存在情况。

# golinter.sh

进行语法检查等，目前主要用 goimports 来检查引入包的语法格式。

# infiniteloop.sh

循环来保持容器始终不退出。

# install_behave.sh

# test

用于测试的一些脚本。

# chaincodes

# AuctionApp

## art.go

## image_proc_api.go

## table_api.go

# BadImport

## main.go

# envsetup

# channel-artifacts

# docker-compose.yaml

# generateCfgTrx.sh

# feature

# configs

## configtx.yaml

## crypto.yaml

# docker-compose

## docker-compose-kafka.yml

## docker-compose-solo.yml

## steps

## basic_impl.py

## compose_util.py

## config_util.py

## endorser_impl.py

## endorser_util.py

## orderer_impl.py

## orderer_util.py

# bootstrap.feature

# environment.py

# orderer.feature

# peer.feature

# regression

# daily

## chaincodeTests

## Example.py

# README.rst.orig

## SampleScriptFailTest.sh

## SampleScriptPassTest.sh

## TestPlaceholder.sh

## ledger_lte.py

## runDailyTestSuite.sh

## systest_pte.py

## testAuctionChaincode.py

## release 包

# byfn_release_tests.py

# e2e_sdk_release_tests.py

# make_targets_release_tests.py

## runReleaseTestSuite.sh

# run_byfn_cli_release_tests.sh

# run_e2e_java_sdk.sh

## run_e2e_node_sdk.sh

## run_make_targets.sh

# run_node_sdk_byfn.sh

## weekly

## runGroup1.sh

## runGroup2.sh

## runGroup3.sh

## runGroup4.sh

## systest_pte.py

## testAuctionChaincode.py

# tools

# AuctionApp

## api_driver.sh

# LTE

## chainmgmt

## common

## experiments

## scripts

# OTE

# PTE

## SCFiles

## userInputs

## chaincode_sample.go

## pte-execRequest.js

## pte-main.js

## pte-util.js

## pte_driver.sh

# docker-compose.yml

启动一个 order 和 一个 peer 节点。

```
orderer:
  image: hyperledger/fabric-orderer
  environment:
    - ORDERER_GENERAL_LEDGERTYPE=ram
    - ORDERER_GENERAL_BATCHTIMEOUT=10s
    - ORDERER_GENERAL_BATCHSIZE=10
    - ORDERER_GENERAL_MAXWINDOWSIZE=1000
    - ORDERER_GENERAL_ORDERERTYPE=solo
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_LISTENPORT=7050
    - ORDERER_RAMLEDGER_HISTORY_SIZE=100
  expose:
    - 7050

vp:
  image: hyperledger/fabric-peer
  links:
    - orderer
  ports:
    - 7051:7051
    - 7053:7053
    - 7054:7054
  environment:
    - CORE_PEER_ADDRESSAUTODETECT=true
    - CORE_PEER_COMMITTER_LEDGER_ORDERER=orderer:7050
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
```

# unit-test

# docker-compose.yml

# run.sh